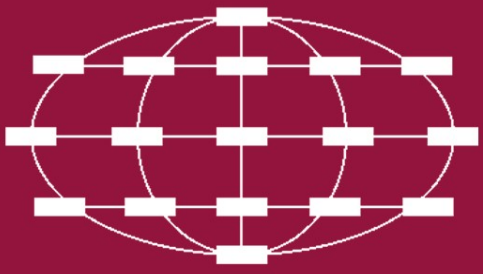
Victor Malyshkin (Ed.)

LNCS 6873

# Parallel Computing Technologies

11th International Conference, PaCT 2011
Kazan, Russia, September 2011
Proceedings



Springer

Lecture Notes in Computer Science 6873

*Commenced Publication in 1973*
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Victor Malyshkin (Ed.)

# Parallel Computing Technologies

11th International Conference, PaCT 2011
Kazan, Russia, September 19-23, 2011
Proceedings

Springer

Volume Editor

Victor Malyshkin
Institute of Computational Mathematics and Mathematical Geophysics
Supercomputer Software Department, Russian Academy of Sciences
Pr. Lavrentieva, ICM&MG RAS, 630090 Novosibirsk, Russia
E-mail: malysh@ssd.sscc.ru

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-23177-3 e-ISBN 978-3-642-23178-0
DOI 10.1007/978-3-642-23178-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011934283

CR Subject Classification (1998): D.2, D.3.2, F.1.2, G.1, G.4, I.6.8, C.1.4, C.2.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues



*Typesetting:* Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Preface

The PaCT 2011 (Parallel Computing Technologies) conference was a four-day conference held in Kazan. This was the 11th international conference in the PaCT series. The conferences are held in Russia every odd year. The first conference, PaCT 1991, was held in Novosibirsk (Academgorodok), September 7-11, 1991. The next PaCT conferences were held in Obninsk (near Moscow), August 30-September 4, 1993; in St. Petersburg, September 12-15, 1995; in Yaroslavl, September, 9-12 1997; in Pushkin (near St. Petersburg), September, 6-10, 1999; in Academgorodok (Novosibirsk), September 3-7, 2001; in Nizhni Novgorod, September, 15-19, 2003; in Krasnoyarsk, September 5-9, 2005; in Pereslavl-Zalessky, September 3-7, 2007; in Novosibirsk, August 31-September 4, 2009. Since 1995 all the PaCT proceedings have been published by Springer in the LNCS series. PaCT 2011 was jointly organized by the Institute of Computational Mathematics and Mathematical Geophysics of the Russian Academy of Sciences (RAS), Institute of Informatics (Academy of Sciences of the Republic of Tatarstan), and Kazan Federal University. The purpose of the conference was to bring together scientists working on theory, architecture, software, hardware and the solution of large-scale problems in order to provide integrated discussions on parallel computing technologies. The conference attracted about 150 participants from around the world. Authors from 13 countries submitted 68 papers. Of those submitted, 44 papers were selected for the conference as regular ones; there were also two invited papers. All the papers were reviewed by at least three international referees. A demo session was organized for the participants. Different tools were submitted for demonstration and tutorials. One of them is WinAlt (Windows Animated Language Tool). Many thanks to our sponsors: Russian Academy of Sciences, Kazan Federal University, Academy of Sciences of the Republic of Tatarstan, Russian Fund for Basic Research, Lufthansa Official Airlines.

September 2011 Victor Malyshkin

# Organization

PaCT 2011 was organized by the Supercomputer Software Department, Institute of Computational Mathematics and Mathematical Geophysics, Siberian Branch, Russian Academy of Science (SB RAS) in cooperation with the Institute of Informatics (Academy of Sciences of the Republic of Tatarstan) and Kazan Federal University.

## Organizing Committee

| | |
|---|---|
| Conference Co-chairs: | Farid Ablayev (Kazan State University, Russia)<br>Victor Malyshkin (Russian Academy of Sciences) |
| Conference Secretariat: | Maxim Gorodnichev<br>Alexander Vasiliev<br>Olga Bandman, Publication Chair<br>Mikhail Abramskiy<br>Ramil Garaev<br>Konstantin Kalgin<br>Sergey Kireev<br>Yulia Kochneva<br>Regina Kozlova<br>Valentina Markova<br>Georgy Schukin<br>Mansur Ziatdinov<br>Alia Zinurova |

## Program Committee

| | |
|---|---|
| V. Malyshkin | Russian Academy of Sciences, Chair |
| F. Ablayev | Kazan Federal University, Russia |
| S. Abramov | Russian Academy of Sciences |
| F. Arbab | Leiden University and CWI, The Netherlands |
| S. Bandini | University of Milan, Italy |
| O. Bandman | Russian Academy of Sciences |
| T. Casavant | University of Iowa, USA |
| P. Degano | State University of Pisa, Italy |
| D. Désérable | Institut National des Sciences Appliqués, France |
| S. Gorlatch | University of Münster, Germany |
| Y. Karpov | St. Petersburg State Technical University, Russia |
| A. Lastovetsky | University College Dublin, Ireland |

| | |
|---|---|
| T. Ludwig | University of Hamburg, Germany |
| G. Mauri | University of Milan, Italy |
| N. Mirenkov | University of Aizu, Japan |
| D. Petcu | Western University of Timisoara, Romania |
| V. Prasanna | University of Southern California, USA |
| M. Raynal | IRISA, Rennes, France |
| B. Roux | IRPHE, France |
| M. Sato | University of Tsukuba, Japan |
| C. Trinitis | LRR, München, Germany; University of Bedfordshire, UK |
| M. Valero | Barcelona Supercomputer Center, Spain |
| R. Wyrzykowski | Czestochowa University of Technology, Poland |
| L. Yang | St. Francis Xavier University, Canada |

## Referees

F. Ablayev
S. Achasova
O. Bandman
O. Bessonov
T. Casavant
A. Cisternino
P. Degano
D. Désérable
G.-L. Ferrari
A. Frangioni
S. Gorlatch
M. Gorodnichev
H. Jeon
K. Kalgin
Yu. Karpov
P. Kegel
S. Kireev
I. Kulikov
A. Lastovetsky
T. Ludwig
N. Ma
V. Malyshkin
M. Marchenko
V. Markova
G. Mauri
Yu. Medvedev
D. Meilaender
A. Nepomniaschaya
M. Ostapkevich
V. Perepelkin
D. Petcu
S. Piskunov
A. Ploss
V. Prasanna
A. Ramírez
M. Raynal
F. Romani
M. Sato
M.G. Scutella
N. Shilov
M. Steuwer
P. Trifonov
C. Trinitis
M. Valero
A. Vasiliev
M. Walter
J. Weidendorfer
R. Wyrzykowski
L. Yang
M. Ziatdinov

## Sponsoring Institutions

Russian Academy of Sciences
Academy of Sciences of the Republic of Tatarstan
Russian Fund for Basic Research
Lufthansa Official Airlines
Kazan Federal University

# Table of Contents

## Models and Languages

Classical and Quantum Parallelism in the Quantum Fingerprinting Method . . . . . 1
*Farid Ablayev and Alexander Vasiliev*

OpenMP Parallelization of a CFD Code for Multicore Computers: Analysis and Comparison . . . . . 13
*Oleg Bessonov*

On Quantitative Security Policies . . . . . 23
*Pierpaolo Degano, Gian-Luigi Ferrari, and Gianluca Mezzetti*

A Formal Programming Model of Orléans Skeleton Library . . . . . 40
*Noman Javed and Frédéric Loulergue*

LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem . . . . . 53
*Victor E. Malyshkin and Vladislav A. Perepelkin*

Grid Computing for Sensitivity Analysis of Stochastic Biological Models . . . . . 62
*Ivan Merelli, Dario Pescini, Ettore Mosca, Paolo Cazzaniga, Carlo Maj, Giancarlo Mauri, and Luciano Milanesi*

Looking for Efficient Implementations of Concurrent Objects . . . . . 74
*Achour Mostéfaoui and Michel Raynal*

Cache Efficiency and Scalability on Multi-core Architectures . . . . . 88
*Thomas Müller, Carsten Trinitis, and Jasmin Smajic*

Symbolic Algorithm for Generation Büchi Automata from LTL Formulas . . . . . 98
*Irina V. Shoshmina and Alexey B. Belyaev*

Sisal 3.2 Language Features Overview . . . . . 110
*Alexander Stasenko*

## Cellular Automata

A Cellular Automata Based Model for Pedestrian and Group Dynamics: Motivations and First Experiments . . . . . 125
*Stefania Bandini, Federico Rubagotti, Giuseppe Vizzari, and Kenichiro Shimura*

Using Multi Core Computers for Implementing Cellular Automata Systems ..... 140
*Olga Bandman*

Efficient Minimal Routing in the Triangular Grid with Six Channels .... 152
*Rolf Hoffmann and Dominique Désérable*

Domain Specific Language and Translator for Cellular Automata Models of Physico-Chemical Processes ..... 166
*Konstantin Kalgin*

Dynamic Load Balancing for Lattice Gas Simulations on a Cluster ..... 175
*Yuri Medvedev*

Efficient Associative Algorithm for Finding the Second Simple Shortest Paths in a Digraph ..... 182
*Anna Nepomniaschaya*

The Construction of Simulation Models of Algorithms and Structures with Fine-Grain Parallelism in WinALT ..... 192
*Mike Ostapkevich and Sergey Piskunov*

Simulation of Heterogeneous Catalytic Reaction by Asynchronous Cellular Automata on Multicomputer ..... 204
*Anastasia Sharifulina and Vladimir Elokhin*

Smallest Implementations of Optimum-Time Firing Squad Synchronization Algorithms for One-Bit-Communication Cellular Automata ..... 210
*Hiroshi Umeo and Takashi Yanagihara*

## Parallel Programming Tools and Support

Distributed Genetic Process Mining Using Sampling ..... 224
*Carmen Bratosin, Natalia Sidorova, and Wil van der Aalst*

FADE: RESTful Service for Failure Detection in SOA Environment ..... 238
*Jerzy Brzeziński, Dariusz Dwornikowski, and Jacek Kobusiński*

ReServE Service: An Approach to Increase Reliability in Service Oriented Systems ..... 244
*Arkadiusz Danilecki, Mateusz Hołenko, Anna Kobusińska, Michał Szychowiak, and Piotr Zierhoffer*

Hypergraph Partitioning for the Parallel Computation of Continuous Petri Nets ..... 257
*Zuohua Ding, Hui Shen, and Jianwen Cao*

High-Performance Reconfigurable Computer Systems ..... 272
*Alexey Dordopulo, Igor Kalyaev, Ilya Levin, and Liubov Slasten*

CacheVisor: A Toolset for Visualizing Shared Caches in Multicore and Multithreaded Processors ..... 284
*Dmitry Evtyushkin, Peter Panfilov, and Dmitry Ponomarev*

The LuNA Library of Parallel Numerical Fragmented Subroutines ..... 290
*Sergey Kireev, Victor Malyshkin, and Hamido Fujita*

PARMONC - A Software Library for Massively Parallel Stochastic Simulation ..... 302
*Mikhail Marchenko*

On Performance Analysis of a Multithreaded Application Parallelized by Different Programming Models Using Intel VTune ..... 317
*Ami Marowka*

Using Multidimensional Solvers for Optimal Data Partitioning on Dedicated Heterogeneous HPC Platforms ..... 332
*Vladimir Rychkov, David Clarke, and Alexey Lastovetsky*

An Efficient Evolutionary Scheduling Algorithm for Parallel Job Model in Grid Environment ..... 347
*Piotr Switalski and Franciszek Seredynski*

On Mapping Graphs of Parallel Programs onto Graphs of Distributed Computer Systems by Recurrent Neural Networks ..... 358
*Mikhail S. Tarkov*

Slot Selection and Co-allocation for Economic Scheduling in Distributed Computing ..... 368
*Victor Toporkov, Alexander Bobchenkov, Anna Toporkova, Alexey Tselishchev, and Dmitry Yemelyanov*

An Initial Approximation to the Resource-Optimal Checkpoint Interval ..... 384
*Ekaterina Tyutlyaeva and Alexander Moskovsky*

## Applications

Performance Characteristics of Global High-Resolution Ocean (MPIOM) and Atmosphere (ECHAM6) Models on Large-Scale Multicore Cluster ..... 390
*Panagiotis Adamidis, Irina Fast, and Thomas Ludwig*

Performances of Navier-Stokes Solver on a Hybrid CPU/GPU Computing System . . . . . 404
*Giancarlo Alfonsi, Stefania A. Ciliberti, Marco Mancini, and Leonardo Primavera*

Optimal Design of Multi-product Batch Plants Using a Parallel Branch-and-Bound Method . . . . . 417
*Andrey Borisenko, Philipp Kegel, and Sergei Gorlatch*

Virtual Path Implementation of Multi-stream Routing in Network on Chip . . . . . 431
*Bartosz Chojnacki, Tomasz Maka, and Piotr Dziurzanski*

Web Service of Access to Computing Resources of BOINC Based Desktop Grid . . . . . 437
*Evgeny Ivashko and Natalia Nikitina*

Solution of Assimilation Observation Data Problem for Shallow Water Equations for SMP-Nodes Cluster . . . . . 444
*Evgeniya Karepova and Ekaterina Dementyeva*

A Parallel Implementation of GaussSieve for the Shortest Vector Problem in Lattices . . . . . 452
*Benjamin Milde and Michael Schneider*

Graphical Algorithm for the Knapsack Problems . . . . . 459
*Alexander Lazarev, Anton Salnikov, and Anton Baranov*

SSCCIP – A Framework for Building Distributed High-Performance Image Processing Technologies . . . . . 467
*Evgeny V. Rusin*

Parallel Logical Cryptanalysis of the Generator A5/1 in BNB-Grid System . . . . . 473
*Alexander Semenov, Oleg Zaikin, Dmitry Bespalov, and Mikhail Posypkin*

High Performance Computing of MSSG with Ultra High Resolution . . . . 484
*Keiko Takahashi, Koji Goto, Hiromitsu Fuchigami, Ryo Onishi, Yuya Baba, Shinichiro Kida, and Takeshi Sugimura*

**Author Index** . . . . . 499

# Classical and Quantum Parallelism in the Quantum Fingerprinting Method

Farid Ablayev[1,2] and Alexander Vasiliev[1,2]

[1] Institute for Informatics, Kazan, Russian Federation
[2] Kazan Federal University, Kazan, Russian Federation

**Abstract.** In this paper we focus on how the classical and quantum parallelism are combined in the quantum fingerprinting technique we proposed earlier. We also show that our method can be used not only to efficiently compute Boolean functions with linear polynomial presentations but also can be adapted for the functions with nonlinear presentations of bounded "nonlinearity".

## 1 Introduction

Nowadays computer science has a strong focus on parallel computing technologies and much effort is being put into parallelizing computational algorithms. On the other hand, there is another promising approach for speeding up computations – the theory of quantum computations, whose power is based on so-called *quantum parallelism*. This effect can be described as evaluating the function for multiple inputs simultaneously, by exploiting the ability of a quantum register to be in superpositions of different states [10]. In some sense the same quantum bits store the results of different evaluations of a function which are performed independently in different subspaces of a state space. Informally, computing classically in parallel means "at the same time", while the quantum parallelism means "using the same space", which also means there is no need in time synchronization for quantum parallelism.

Nevertheless, classical parallelization is also extremely important for quantum information processing, since the decoherence in quantum computers limits the number of computational steps. Thus, the classical and quantum parallelism should be used to complement each other in order to perform high-productivity computations (in both time and space). In this paper we exhibit how this is done in the generalized quantum fingerprinting technique presented in [2].

There are many models of computation based on the quantum paradigm. These models are

- discrete deterministic reversible linear models due to their transformations;
- possessing an ability to perform massive parallel computations;
- probabilistic models due to the measurement result extraction.

Due to severe limits of existing physical implementations of quantum computer it is natural to consider the restricted models of quantum computations. The one

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 1–12, 2011.

we consider in this paper is based upon *quantum branching programs*. Two variants of quantum branching programs were introduced by Ablayev, Gainutdinova, Karpinski [1] (*leveled programs*), and by Nakanishi, Hamaguchi, Kashiwabara [9] (*non-leveled programs*). Later it was shown by Sauerhoff [11] that these two models are polynomially equivalent. The most commonly used restricted variant of quantum branching programs is the model of *Ordered Read-Once Quantum Branching Programs*. In computer science this model is also known as Ordered Binary Decision Diagrams (OBDDs). This restriction implies that each input variable may be read at most once, which is the least possible for any function essentially depending on it's variables. Thus, the read-once restriction corresponds to minimizing of computational steps for quantum algorithms.

For this model we develop the *fingerprinting* technique, which is generally used to perform space-efficient computations in randomized and quantum models of computation. At the heart of our technique lies the polynomial presentation of Boolean functions, which we call *characteristic*. The polynomial presentations of Boolean functions are widely used in theoretical computer science. For instance, an algebraic transformation of Boolean functions has been applied in [7] and [4] for verification of Boolean functions. In the quantum setting polynomial representations were used for proving lower bounds on communication complexity in [5] as well as for investigating query complexity in [13]. Our approach combines the ideas similar to the definition of characteristic polynomial from [7], [4] and to the notion of *zero-error polynomial* (see, e.g. [13]).

In [2], [3] we have shown that Boolean functions with linear polynomial presentations can be efficiently computed in the model of quantum OBDDs. In this paper we generalize this result outlining the class of functions with nonlinear presentations that can be efficiently computed via our fingerprinting method.

## 2 Preliminaries

We use the notation $|i\rangle$ for the vector from $\mathcal{H}^d$, which has a 1 on the $i$-th position and 0 elsewhere. An orthonormal basis $|1\rangle,\ldots,|d\rangle$ is usually referred to as the *standard computational basis*. In this paper we consider all quantum transformations and measurements with respect to this basis.

**Definition 1.** *A Quantum Branching Program $Q$ over the Hilbert space $\mathcal{H}^d$ is defined as*

$$Q = \langle T, |\psi_0\rangle, \mathrm{Accept}\rangle \ , \tag{1}$$

*where $T$ is a sequence of $l$ instructions: $T_j = \big(x_{i_j}, U_j(0), U_j(1)\big)$ is determined by the variable $x_{i_j}$ tested on the step $j$, and $U_j(0)$, $U_j(1)$ are unitary transformations in $\mathcal{H}^d$.*

*Vectors $|\psi\rangle \in \mathcal{H}^d$ are called states (state vectors) of $Q$, $|\psi_0\rangle \in \mathcal{H}^d$ is the initial state of $Q$, and* Accept $\subseteq \{1, 2, \ldots d\}$ *is the set of indices of accepting basis states.*

*We define a computation of $Q$ on an input $\sigma = \sigma_1 \ldots \sigma_n \in \{0,1\}^n$ as follows:*

1. *A computation of $Q$ starts from the initial state $|\psi_0\rangle$;*

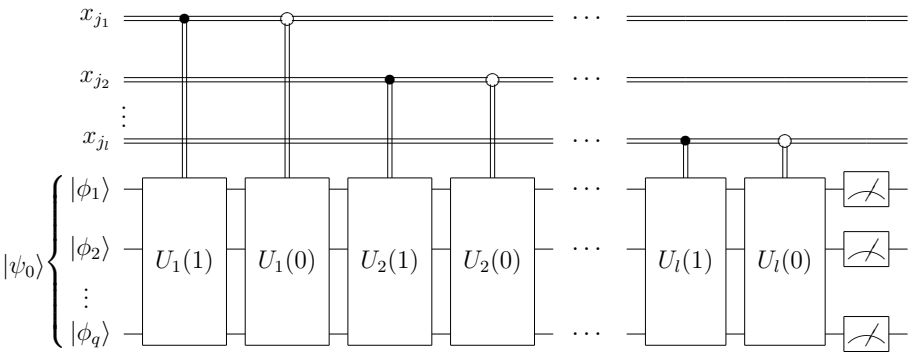



**Fig. 1.** Circuit presentation of a quantum branching program. Here $x_{i_1}, \ldots, x_{i_l}$ is the sequence of (not necessarily distinct) variables denoting classical control (input) bits. Using the common notation single wires carry quantum information and double wires denote classical information and control.

2. *The $j$-th instruction of $Q$ reads the input symbol $\sigma_{i_j}$ (the value of $x_{i_j}$) and applies the transition matrix $U_j = U_j(\sigma_{i_j})$ to the current state $|\psi\rangle$ to obtain the state $|\psi'\rangle = U_j(\sigma_{i_j})\,|\psi\rangle$;*
3. *The final state is*

$$|\psi_\sigma\rangle = \left(\prod_{j=l}^{1} U_j(\sigma_{i_j})\right) |\psi_0\rangle \quad . \tag{2}$$

4. *After the $l$-th (last) step of quantum transformation $Q$ measures its configuration $|\psi_\sigma\rangle = (\alpha_1, \ldots, \alpha_d)^T$, and the input $\sigma$ is accepted with probability*

$$Pr_{\text{accept}}(\sigma) = \sum_{i \in \text{Accept}} |\alpha_i|^2 \quad . \tag{3}$$

*Circuit Representation.* Quantum algorithms are usually given by using quantum circuit formalism [6], [14], because this approach is quite straightforward for describing such algorithms.

We propose, that a QBP represents a classically-controlled quantum system. That is, a QBP can be viewed as a quantum circuit aided with an ability to read classical bits as control variables for unitary operations.

*Example.* As an example consider the Boolean function $MOD_m(x_1, \ldots, x_n)$ which tests whether the number of ones in it's input is a multiple of $m$. For this function the simple algorithm can be proposed (see Figure 2).

The algorithm starts with a qubit in basis state $|0\rangle$. At $j$-th step the value of $x_j$ is tested. Upon input symbol 0 identity transformation $I$ is applied. But if the value of $x_j$ is 1, then the state of the qubit is transformed by the operator $R$, rotating it by the angle proportional to $\pi/m$.

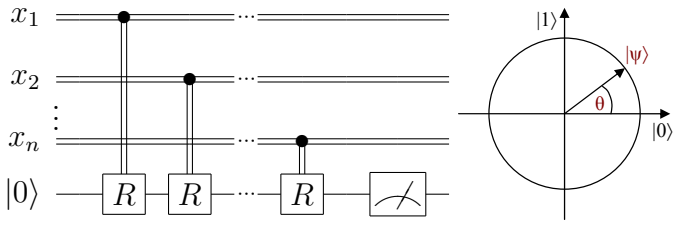



**Fig. 2.** Quantum branching program for $MOD_m$ Boolean function. Here $R$ denotes the rotation by an angle $\theta = \pi/m$ about the $\hat{y}$ axis of the Bloch sphere.

The final state is measured in the standard computational basis. The input $\sigma = \sigma_1 \dots \sigma_n$ is accepted if the result is the basis state $|0\rangle$, otherwise the input $\sigma$ is rejected. For arbitrary input $\sigma$ the acceptance probability equals to

$$Pr_{accept}(\sigma) = \cos^2\left(\frac{\pi \sum_i \sigma_i}{m}\right) . \tag{4}$$

Thus, if $MOD_m(\sigma) = 1$ then $Pr_{accept}(\sigma) = 1$. If $MOD_m(\sigma) = 0$ then the probability of erroneously obtaining the $|0\rangle$ can be close to 1, but this can be improved by applying the fingerprinting techniques described in the next section.

*Complexity Measures.* The width of a QBP $Q$, denoted by width($Q$), is the dimension $d$ of the corresponding state space $\mathcal{H}^d$, and the length of $Q$, denoted by length($Q$), is the number $l$ of instructions in the sequence $T$.

In this paper we're mostly interested in another important complexity for a QBP $Q$ – a number of quantum bits, denoted by qubits($Q$), physically needed to implement a corresponding quantum system with classical control. From definition it follows that $\log \text{width}(Q) \leq \text{qubits}(Q)$.

*Acceptance Criteria.* A QBP $Q$ *computes the Boolean function* $f$ *with bounded error* if there exists an $\epsilon \in (0, 1/2)$ (called *margin*) such that for all inputs the probability of error is bounded by $1/2 - \epsilon$.

In particular, we say that a QBP $Q$ *computes the Boolean function* $f$ *with one-sided error* if there exists an $\epsilon \in (0, 1)$ (called *error*) such that for all $\sigma \in f^{-1}(1)$ the probability of $Q$ accepting $\sigma$ is 1 and for all $\sigma \in f^{-1}(0)$ the probability of $Q$ erroneously accepting $\sigma$ is less than $\epsilon$.

*Read-Once Branching Programs.* Read-once BPs is a well-known restricted variant of branching programs [12].

**Definition 2.** *We call a* QBP $Q$ *a quantum* OBDD (QOBDD) *or read-once* QBP *if each variable* $x \in \{x_1, \dots, x_n\}$ *occurs in the sequence* $T$ *of transformations of* $Q$ *at most once.*

For the rest of the paper we're only interested in QOBDDs, i.e. the length of all programs would be $n$ (the number of input variables).

## 3 Algorithms for QBPs Based on Fingerprinting

Generally [8], *fingerprinting* is a technique that allows to present objects (words over some finite alphabet) by their *fingerprints*, which are significantly smaller than the originals. It is used in randomized and quantum algorithms to test *equality* of some objects (binary strings) with one-sided error by simply comparing their fingerprints.

In the next subsection we show the basic idea of fingerprinting from [2], [3] and compare the usage of classical and quantum parallelism for this method.

### 3.1 Basic Idea

Let $\sigma = \sigma_1 \dots \sigma_n$ be an input string and $g$ is the mapping of $\{0,1\}^n$ onto $\mathbb{Z}_m$ that "encodes" some property of the input we're about to test. We consider $g$ to be the polynomial over $\mathbb{Z}_m$ such that $g(\sigma) = 0 \bmod m \iff \sigma$ has the property encoded by $g$. For example, if we test the equality of two $n$-bit binary strings $x_1 \dots x_n$ and $y_1 \dots y_n$, we can choose $g$ equal to the following polynomial over $\mathbb{Z}_{2^n}$:

$$\sum_{i=1}^{n} x_i 2^{i-1} - \sum_{i=1}^{n} y_i 2^{i-1} \quad . \tag{5}$$

**Trivial Approach.** To test the property encoded by $g$ we rotate the initial state $|0\rangle$ of a single qubit by an angle $\theta = \pi g(\sigma)/m$:

$$|0\rangle \to \cos\theta\,|0\rangle + \sin\theta\,|1\rangle \quad . \tag{6}$$

Obviously, this quantum state is exactly $|0\rangle \iff g(\sigma) = 0 \bmod m$. In the worst case this algorithm gives the one-sided error of $\cos^2 \pi(m-1)/m$, which can be arbitrarily close to 1. Figure 3 illustrates the rotated qubit with real amplitudes.

**Classical Parallelism.** To improve this construction we increase the number of qubits and introduce additional parameters. Let $k_1, \dots, k_t \in \{1, \dots, m-1\}$. Then we rotate $t$ isolated qubits in the state $|0\rangle$ by angles $\theta_i = \pi k_i g(\sigma)/m$ (see Figure 4):

$$|0\rangle \to \cos\theta_i\,|0\rangle + \sin\theta_i\,|1\rangle \quad . \tag{7}$$

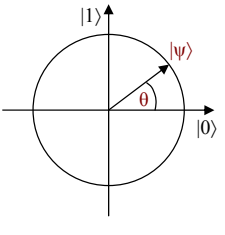



**Fig. 3.** A single qubit rotated by an angle $\theta$

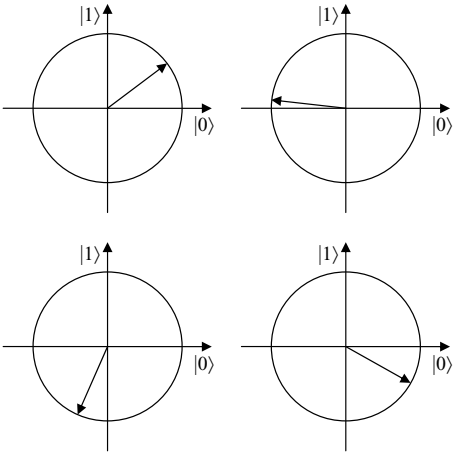



**Fig. 4.** $t$ single qubits rotated by an angle $\theta_i$

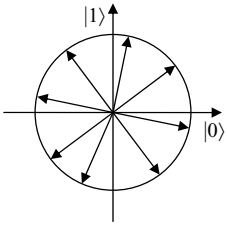



**Fig. 5.** A qubit rotated in parallel by $t$ different angles $\theta_i$

If we measure these qubits we obtain the state $|0\rangle$ of $i$-th qubit with probability $\cos^2 \frac{\pi k_i g(\sigma)}{m}$, which is 1 when $g(\sigma)$ equals 0 modulo $m$.

On the other hand there exists a set $K = \{k_1, \ldots, k_t\}$ with $t = O(\log m)$, such that $Pr_{\text{error}} < 1/m$ for all $\sigma$ with $g(\sigma) \neq 0 \bmod m$.

**Quantum Parallelism.** Using quantum effects like *entanglement*, *quantum parallelism* and *interference* [10], we can decrease the number of qubits to $\log t + 1$:

$$\underbrace{|0\rangle \otimes |0\rangle \otimes \cdots \otimes |0\rangle}_{\log t = O(\log\log m)} \otimes |0\rangle \longrightarrow \frac{1}{\sqrt{t}} \sum_{i=1}^{t} |i\rangle \left( \cos\theta_i |0\rangle + \sin\theta_i |1\rangle \right) , \qquad (8)$$

where $\theta_i = \frac{2\pi k_i g(\sigma)}{m}$ and the set $K = \{k_1, \ldots, k_t\}$ is chosen in order to guarantee the small probability of error [2], [3].

That is, the last qubit is simultaneously rotated in $t$ different subspaces by corresponding angles. This approach is informally illustrated by the Figure 5.

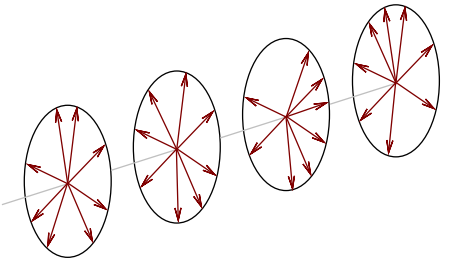


**Fig. 6.** $l$ parallel qubits rotated in parallel by $t$ different angles

**Generalized Approach.** Generalizing this approach, we can combine the classical and quantum parallelism to compute more complex Boolean functions. For illustration of the generalized fingerprinting technique see Figure 6, the detailed description of this approach is given in subsection 3.3.

### 3.2 Quantum Algorithms Based on Fingerprinting

Summarizing, our method may be applied in the following manner:

1. The initial state of the quantum register is $|0\rangle^{\otimes \log t} |0\rangle$.
2. The Hadamard transform creates the equal superposition of the basis states $\frac{1}{\sqrt{t}} \sum_{j=1}^{t} |j\rangle |0\rangle$
3. Based on the input $\sigma$ it's fingerprint is created: $\frac{1}{\sqrt{t}} \sum_{j=1}^{t} |j\rangle \left( \cos \frac{2\pi k_j g(\sigma)}{m} |0\rangle + \sin \frac{2\pi k_j g(\sigma)}{m} |1\rangle \right)$
4. The Hadamard transform turns the fingerprint into the superposition $\left( \frac{1}{t} \sum_{l=1}^{t} \cos \frac{2\pi k_l g(\sigma)}{m} \right) |0\rangle^{\otimes \log t} |0\rangle + \ldots$
5. The quantum register is measured and the input is accepted iff the result is $|0\rangle^{\otimes \log t} |0\rangle$.

### 3.3 Fingerprinting Technique

The fingerprinting technique described in [2], [3] allows us to test the conjunction of several conditions encoded by a group of characteristic polynomials which we call a *characteristic* of a function.

**Definition 3.** *We call a set $\chi_f$ of polynomials over $\mathbb{Z}_m$ a* characteristic *of a Boolean function $f$ if for all polynomials $g \in \chi_f$ and all $\sigma \in \{0,1\}^n$ it holds that $g(\sigma) = 0$ iff $\sigma \in f^{-1}(1)$.*

We use this definition to formulate our fingerprinting technique.

*Fingerprinting Technique.* For a Boolean function $f$ we choose an error rate $\epsilon > 0$ and pick a characteristic $\chi_f = \{g_1, \ldots, g_l\}$ over some ring $\mathbb{Z}_m$. Then for arbitrary binary string $\sigma = \sigma_1 \ldots \sigma_n$ we create its fingerprint $|h_\sigma\rangle$ composing $t \cdot l$ ($t = 2^{\lceil \log((2/\epsilon) \ln 2m) \rceil}$) single qubit fingerprints $\left|h^i_\sigma(j)\right\rangle$

$$\left|h^i_\sigma(j)\right\rangle = \cos \frac{\pi k_i g_j(\sigma)}{m} |0\rangle + \sin \frac{\pi k_i g_j(\sigma)}{m} |1\rangle \tag{9}$$

into entangled state of $\log t + l$ qubits:

$$|h_\sigma\rangle = \frac{1}{\sqrt{t}} \sum_{i=1}^{t} |i\rangle \left|h^i_\sigma(1)\right\rangle \left|h^i_\sigma(2)\right\rangle \ldots \left|h^i_\sigma(l)\right\rangle \quad . \tag{10}$$

Here the transformations of the last $l$ qubits in $t$ different subspaces "simulate" the transformations of all of the $\left|h^i_\sigma(j)\right\rangle$ ($i = 1, \ldots, t, j = 1, \ldots, l$). That is, these $l$ qubits are used in parallel (classically) and each of them is in parallel (quantumly) rotated by $t$ different angles about the $\hat{y}$ axis of the Bloch sphere. Figure 6 informally demonstrates this interpretation of our fingerprinting technique.

The set of parameters $k_i \in \{1, \ldots, m-1\}$ for $i \in \{1, \ldots, t\}$ is chosen in a special way in order to bound the probability of error [2]. That is, it allows to distinguish with high probability those inputs whose image is 0 modulo $m$ from the others.

We say that a characteristic is *linear* if all of its polynomials are linear. In [2] we have shown that Boolean functions with linear characteristics of logarithmic size can be efficiently computed in the quantum OBDD model.

In the next subsection we show how this result can be extended to the case of Boolean functions with nonlinear polynomial presentations.

### 3.4 Computing Functions with Nonlinear Characteristics

In this subsection we describe a class of effectively computable Boolean functions with characteristics of small "nonlinearity", but first we introduce the measure of this quantity.

Let $f(x_1, \ldots, x_n)$ be a Boolean function we are about to compute and $\chi_f$ be it's characteristic over some ring $\mathbb{Z}_m$. Denote $X_n = \{x_1, \ldots, x_n\}$ and let $X \subseteq X_n$ be some subset of variables used by $f$.

**Definition 4.** *We call $\chi_f$ the $X$-nonlinear characteristic of $f$ if $X$ is the minimal set, containing all of the variables that appear in any multilinear term of any polynomial $g \in \chi_f$.*

As a special case of this definition we obtain the notion of linear characteristic, when $X = \emptyset$.

In this notation we can prove the following theorem.

**Theorem 1.** *If $\chi_f$ is an $X$-nonlinear characteristic for Boolean function $f$ then for any $\epsilon \in (0, 1)$ $f$ can be computed with one-sided error $1/2 + \sqrt{\epsilon}/2$ by a quantum OBDD $Q$ with* $\text{qubits}(Q) = O(|\chi_f| + |X| + \log \log m)$.

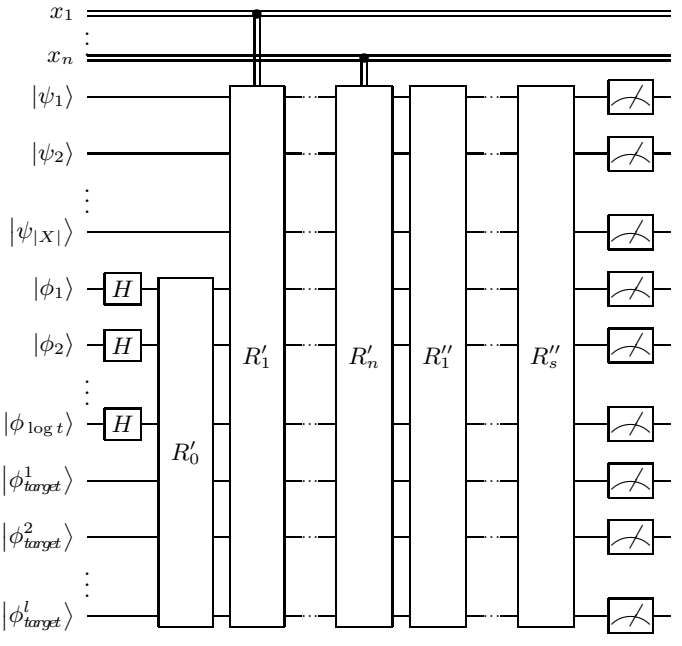



**Fig. 7.** The schematic circuit for computing Boolean function with nonlinear polynomial presentation

*Proof.* The main idea of the algorithm is to save the values of variables in $X$ using $|X|$ qubits and reconstruct the algorithm from [2] so it will use quantum rather than classical control on those values.

Let $\chi_f = \{g_1, g_2, \ldots, g_l\}$ and $g_j = c_0^j + c_1^j x_1 + \ldots + c_n^j x_n + \sum_{r=1}^{s_j} q_r$, where $q_r$ are multilinear terms. The construction of a fingerprint is split in three major steps illustrated in Figures 7, 8, 9, 10:

1. The operator $R'_0$ computes the constant part of all polynomials, that is the terms $c_0^j$.
2. Each operator $R'_i$, corresponding to the $i$-th input variable, computes the linear terms $c_i^j x_i$. Additionally, it saves the value of $x_i$ if it belongs to the set $X$.
3. Finally, for each multilinear term of every polynomial in $\chi_f$ there is an operator $R''_i$ computing it based on the values saved at the previous step.

All of these steps use essentially the same controlled operation $R_i(c) = R_{\hat{y}}\left(\frac{2\pi k_i c}{m}\right)$, which is the rotation about the $\hat{y}$ axis of the Bloch sphere by the corresponding angle.

The computation of the function $f$ itself is as following:

1. Upon the input $\sigma = \sigma_1 \ldots \sigma_n$ we create it's fingerprint $|h_\sigma\rangle$ which has the form of Equation 10.

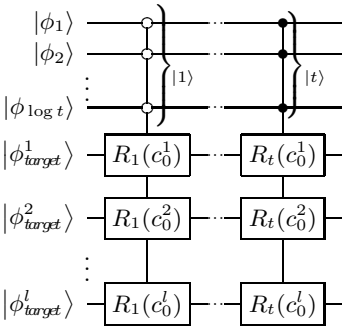



**Fig. 8.** The circuit for operator $R'_0$, computing the constant part ($c^j_0$) of all polynomials

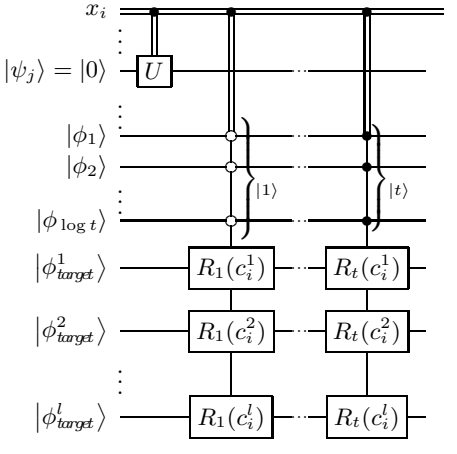



**Fig. 9.** The circuit for operators $R'_i$, computing the linear terms $c^j_i x_i$ of all polynomials. The operator $U$ is the NOT operation if $x_i \in X$ and identity operator otherwise.

2. We measure $|h_\sigma\rangle$ in the standard computational basis and accept the input if the outcome of the last $l$ qubits is the all-zero state. Thus, the probability of accepting $\sigma$ is

$$Pr_{accept}(\sigma) = \frac{1}{t}\sum_{i=1}^{t} \cos^2 \frac{\pi k_i g_1(\sigma)}{m} \cdots \cos^2 \frac{\pi k_i g_l(\sigma)}{m}.$$

If $f(\sigma) = 1$ then all of $g_i(\sigma) = 0$ and we will always accept.
If $f(\sigma) = 0$ then there is at least one such $j$ that $g_j(\sigma) \neq 0$ and the choice of the set $K$ guarantees [2] that the probability of the erroneously accepting is bounded by

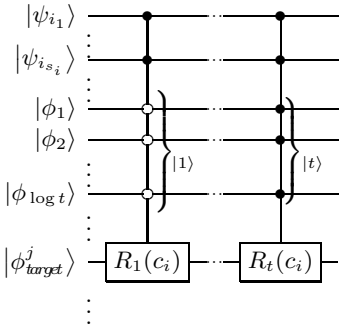



**Fig. 10.** The circuit for operators $R_i''$, computing the multilinear term $c_i x_{i_1} x_{i_2} \cdots x_{i_{s_i}}$ of the $j$-th polynomial

$$\begin{aligned} Pr_{accept}(\sigma) &= \tfrac{1}{t} \sum_{i=1}^{t} \cos^2 \tfrac{\pi k_i g_1(\sigma)}{m} \cdots \cos^2 \tfrac{\pi k_i g_l(\sigma)}{m} \\ &\le \tfrac{1}{t} \sum_{i=1}^{t} \cos^2 \tfrac{\pi k_i g_j(\sigma)}{m} = \tfrac{1}{t} \sum_{i=1}^{t} \tfrac{1}{2} \left(1 + \cos \tfrac{2\pi k_i g_j(\sigma)}{m}\right) \\ &= \tfrac{1}{2} + \tfrac{1}{2t} \sum_{i=1}^{t} \cos \tfrac{2\pi k_i g_j(\sigma)}{m} \\ &\le \tfrac{1}{2} + \tfrac{\sqrt{\epsilon}}{2}. \end{aligned}$$

The number of qubits used by this QBP $Q$ is $qubits(Q) = O(\log \log m + |\chi_f| + |X|)$. □

As an immediate corollary we obtain that we can efficiently compute any Boolean function $f$ with $X$-nonlinear characteristic $\chi_f$ over $\mathbb{Z}_m$, whenever $|X| = |\chi_f| = O(\log n)$ and $m = 2^{n^{O(1)}}$.

**Acknowledgments.** Work was in part supported by the Russian Foundation for Basic Research under the grants 09-01-97004 and 11-07-00465.

## References

1. Ablayev, F., Gainutdinova, A., Karpinski, M.: On computational power of quantum branching programs. In: Freivalds, R. (ed.) FCT 2001. LNCS, vol. 2138, pp. 59–70. Springer, Heidelberg (2001), http://arxiv.org/abs/quant-ph/0302022
2. Ablayev, F., Vasiliev, A.: Algorithms for quantum branching programs based on fingerprinting. EPTCS 9, 1–11 (2009), http://arxiv.org/abs/0911.2317
3. Ablayev, F., Vasiliev, A.: On computational power of quantum read-once branching programs. Electronic Proceedings in Theoretical Computer Science 52, 1–12 (2011), http://arxiv.org/abs/1103.2809v1
4. Agrawal, V., Lee, D., Wozniakowski, H.: Numerical computation of characteristic polynomials of boolean functions and its applications. Numerical Algorithms 17, 261–278 (1998), http://dx.doi.org/10.1023/A:1016632423579

5. Buhrman, H., Cleve, R., Watrous, J., de Wolf, R.: Quantum fingerprinting. Phys. Rev. Lett. 87(16), 167902 (2001), www.arXiv.org/quant-ph/0102001v1
6. Deutsch, D.: Quantum computational networks. Royal Society of London Proceedings Series A 425, 73–90 (1989), http://dx.doi.org/10.1098/rspa.1989.0099
7. Jain, J., Abraham, J.A., Bitner, J., Fussell, D.S.: Probabilistic verification of boolean functions. Formal Methods in System Design 1, 61–115 (1992)
8. Motwani, R., Raghavan, P.: Randomized algorithms. Cambridge University Press, Cambridge (1995)
9. Nakanishi, M., Hamaguchi, K., Kashiwabara, T.: Ordered quantum branching programs are more powerful than ordered probabilistic branching programs under a bounded-width restriction. In: Du, D.-Z., Eades, P., Sharma, A.K., Lin, X., Estivill-Castro, V. (eds.) COCOON 2000. LNCS, vol. 1858, pp. 467–476. Springer, Heidelberg (2000)
10. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information, 1st edn. Cambridge University Press, Cambridge (2000), http://www.worldcat.org/isbn/521635039
11. Sauerhoff, M., Sieling, D.: Quantum branching programs and space-bounded nonuniform quantum complexity. Theoretical Computer Science 334(1-3), 177–225 (2005), http://arxiv.org/abs/quant-ph/0403164
12. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM Monographs on Discrete Mathematics and Applications. SIAM Press, Philadelphia (2000)
13. de Wolf, R.: Quantum Computing and Communication Complexity. Ph.D. thesis, University of Amsterdam (2001)
14. Yao, A.C.C.: Quantum circuit complexity. In: Proceedings of Thirty-fourth IEEE Symposium on Foundations of Computer Science, pp. 352–361. IEEE Computer Society, Palo Alto (1993)

# OpenMP Parallelization of a CFD Code for Multicore Computers: Analysis and Comparison

Oleg Bessonov

Institute for Problems in Mechanics of the Russian Academy of Sciences
101, Vernadsky ave., 119526 Moscow, Russia
bess@ipmnet.ru

**Abstract.** In this paper we present a parallelization method for numerical simulation of incompressible flows in regular domains on multicore computers in frame of the OpenMP programming model. The method is based on natural splitting of a computational domain for the main part of the algorithm, and on two-dimensional splitting and application of a special tridiagonal parallelization procedure for pressure Poisson equation and other implicit parts. This method is suitable for running on shared memory computer systems with non-uniform memory and demonstrates good parallelization efficiency for up to 16 threads.

## 1 Introduction

It is well known that numerical simulations of convective interactions and instabilities in crystal growth applications require huge computational resources [1]. For example, modeling of oscillating behaviour of non-axisymmetric flow in a cylindrical domain with one million grid points for one million time steps takes more than 10 days on a single-core processor of 2004-year generation [2]. Serial performance of modern microprocessors is only 2 to 3 times higher, therefore parallelization is required for further acceleration of the simulation process.

Previously, only distributed memory computer systems (clusters) were generally available for parallel computations. Such systems were built on computer nodes having usually two single-core processors. Thus, the only practical way of parallelization was the MPI distributed-memory approach. Significant experience was acquired with MPI parallelization of codes for simulating incompressible flows [3,4].

Currently multicore processors become widely available. Typical processor has now 4 to 6 cores and looks as a high-performance shared memory computer system. A small bi-processor server (or cluster node) can have 8 to 12 cores. Because of wide availability of multicore processor systems, performance criteria and methods have now to be revisited, together with reconsideration of parallelization. The latter becomes important because now parallelization is no more an option as before. Currently it is a must because there is no other way to fully utilize the computational potential of a processor. With parallelization, an inexpensive quad-core desktop computer in 2010 becomes about 10 times faster than a typical computer in 2005.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 13–22, 2011.

Changing the sort of target computer systems and parallelization criteria should result in changing a parallelization approach. For shared memory systems, the OpenMP model is more simple and natural [5]. This model can be used for developing new computational codes, as well as for smooth conversion of existing ones. At last, it ensures higher parallel efficiency than the less natural MPI approach.

Modern shared memory computers can be subdivided into several classes: single-processor desktop systems, bi-processor servers (nodes) and more expensive multiprocessor servers. Multiprocessor and (partly) bi-processor systems possess the property of non-uniform memory organization (NuMA). Unlike traditional computers with uniform memory, NuMA systems need special programming of parallel applications because of requirement for each thread to access only (mostly) data resided in a processor's local memory. The NuMA-specific approach to OpenMP programming was considered in the previous work [6] devoted to the parallelization of a full-physical model for forest fire simulation.

The present work combines and extends the previous experience [4,6]. Its goal is to develop and analyze the OpenMP approach for modeling of incompressible flows in regular domains. Target computer systems for this investigation are desktops and small servers (cluster nodes) having up to 12-16 processor cores. These values determine the reasonable level of parallelization and, therefore, selection of a parallelization method.

Thereby, in the presented paper we will describe mathematical model and numerical method, strategy of OpenMP parallelization and comparison of parallel efficiency for several types of computers systems.

## 2 Numerical Method

A numerical problem considered in this paper is the solution of 3D non-stationary Navier-Stokes equations in Boussinesq approximation for incompressible viscous flows in cylindrical domains. This sort of simulation is used in crystal growth applications, like semiconductor melt flows in a Czochralski apparatus, and in modeling of natural convection in space experiments. The velocity-pressure formulation is employed, with the decoupled solution of momentum ($\boldsymbol{V}$), pressure ($p$) and temperature ($\theta$) equations using the Fractional step method:

$$\frac{\partial \boldsymbol{V}}{\partial t} + \boldsymbol{\nabla} \cdot (\boldsymbol{V}\boldsymbol{V}) = -\boldsymbol{\nabla} p + \nabla^2 \boldsymbol{V} - \frac{\mathrm{Ra}}{\mathrm{Pr}} \boldsymbol{g}\, \theta$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{V} = 0$$

$$\frac{\partial \theta}{\partial t} + \boldsymbol{\nabla} \cdot (\boldsymbol{V}\theta) = \frac{1}{\mathrm{Pr}} \nabla^2 \theta$$

The Finite Volume method (FVM) discretization is applied to the regular staggered grid, with accurate treatment of the cylinder axis for essentially non-axisymmetric flows. Second-order spatial discretizations are used, with the quadratic upstream interpolation of advective terms (QUICK scheme) and optional flux limiter for the scalar transport with high Peclet number.

The time integration scheme is partially implicit, with the implicit treatment of the most critical terms. The pressure Poisson equation is solved by the novel highly efficient direct method based on the decomposition of a matrix on the basis of its eigenfunctions (considered below). Using this new method instead of the traditional Fourier approach is necessary in order to apply a highly nonuniform grid in the axial direction for the simulation of convective processes in very thin boundary layers (e.g. near the surface of a growing crystal).

This numerical method is fully direct and doesn't involve costly iterative steps. Therefore it is applicable for accurate and efficient simulations of transitional and turbulent flows with the good spatial and temporal resolution.

Figure 1 shows a cylindrical computational domain for Czochralski hydrodynamic model (left) and examples of isolines and trajectories in a typical non-axisymmetric flow simulation (right).

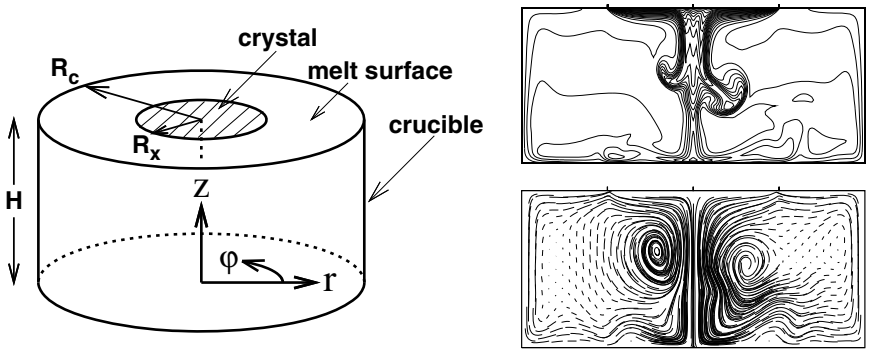



**Fig. 1.** Computational model of a Czochralski apparatus (left) and examples of simulated flow pictures (right)

## 3 Solving the Pressure Poisson Equation

For the solution of the pressure Poisson equation, a new direct method was developed. This method is based on the concept of separation of variables.

The Laplace operator possesses the property of separability, when each directional part depends only on one coordinate, e.g. $L(i,j) = L_{xx}(i) + L_{yy}(j)$ (2D case). This property can be used as a basis for constructing the method of separation of variables. An elliptic equation with a separable operator looks as

$$L_{xx}(i)q(i,j) + L_{yy}(j)q(i,j) = f(i,j)$$

We will find the solution of this equation using the series expansion on the basis of eigenfunctions of the first part of the Laplace operator:

$$q(i,j) = \sum_m \tilde{q}_m(j)\varphi_m(i)$$

$$f(i,j) = \sum_m \tilde{f}_m(j)\varphi_m(i)$$

Each eigenfunction $\varphi_m(i)$ has its own eigenvalue $\lambda_m$ such that:

$$L_{xx}(i)\varphi_m(i) = \lambda_m \varphi_m(i)$$

Since all eigenfunctions are orthogonal, the original discrete 2D equation can be transformed into a set of independent 1D equations:

$$\lambda_m \tilde{q}_m(j) + L_{yy}(j)\tilde{q}_m(j) = \tilde{f}_m(j)$$

This technique is called a matrix decomposition approach and is used for reducing the number of dimensions in a separable linear system. The algorithm consists of 3 steps: expansion of the function $f(i, j)$ from the right hand side of a 2D equation into the series $\tilde{f}_m(j)$, solution of independent 1D equations, and synthesis of the unknown $q(i, j)$ from the series $\tilde{q}_m(j)$. Both expansion (analysis) and synthesis can be performed by matrix multiplication, using the matrix of eigenfunctions and its inverse (prepared preliminarily in the beginning of a run).

The well-known Fourier method for solving the Poisson equation is a particular case of the general matrix decomposition technique. In this method, very efficient Fast Fourier transform (FFT) is used instead of more costly matrix multiplication. However, the Fourier method imposes severe restrictions on grid size and uniformity. As a result, it can't be used in more general cases.

In 3D case, it is possible to apply the matrix decomposition technique twice in order to obtain finally a set of separate 1D equations. In the current cylindrical coordinate approach, an FFT is applied first for the azimuthal direction. Then, a general matrix decomposition is used for the axial direction. Finally, resulting tridiagonal linear systems are solved for the remaining radial direction.

Formally, the new method is much more expensive than other direct methods. Its cost is $O(N)$ operations per grid point, where $N$ is the corresponding dimension. In comparison, operation count for the Fourier method is only $O(\log(N))$. Another competitive direct method, 1-way dissection (1-WD), that is based on the splitting of a region into narrow stripes and renumbering of grid nodes, costs $O(\text{sqrt}(N))$. However, matrix multiplications (as a main part of the new method) can be executed very fast on modern processors: highly optimized procedures of BLAS-3 library achieve up to 80% of peak processor performance. In contrast, FFT and 1-WD methods have very complicated memory access patterns and are unable to run as fast.

As a result, the new method becomes very competitive: it is only about 1.5 times more expensive than FFT. At the same time it doesn't impose restrictions on the grid uniformity. Additionally, the method of separation of variables is more suitable for parallelization than another non-restrictive direct methods (like 1-WD) because it consists in solving independent 1D equations and doesn't require to discover parallelization potential as in the case of multidimensional linear solvers (example of a similar problem is considered in [7]).

It should be noted that direct methods are generally much more efficient for solving Poisson equations than iterative ones (though more restrictive). For example, the method above works on SGI Altix computer at least 10 times faster (depending on grid size and parallelization level) than the advanced LU-preconditioned Conjugate gradient solver [7].

## 4 Parallelization Method

OpenMP approach [5] used in the current work is very convenient for portable parallelization of many algorithms. If a computational domain is of regular shape (cylinder, rectangular parallelepiped) then the natural splitting can be used: each 3D array being processed throughout the algorithm is divided by the last spacial dimension (last index in Fortran notation). This splitting is performed automatically, a user should only apply an `!$OMP DO` directive to each outermost do-loop concerned [6]. Splitting by outermost loop iterations corresponds to one-dimensional geometric decomposition of a computational domain (Fig. 2, left). This decomposition is natural and efficient with respect to the requirements of data distribution for systems with hierarchical memory organization. Additionally, one-dimensional splitting allows to fulfil the requirement for non-uniform memory (NuMA) computers that all (most) data accessed by a processor must reside in its local memory. The easiest way to achieve this is to initialize all data arrays in parallel loops with the same splitting as processing loops. In this case, physical pages of data arrays are allocated in corresponding local memories [6].

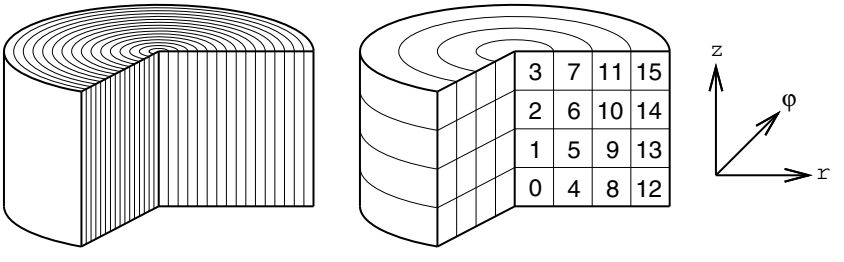



**Fig. 2.** Splitting of a computational domain for simple (non-algebraic) and complex (algebraic) parts of the algorithm

However, automatic OpenMP parallelization can be applied only if there are no recursive dependences between outer loop iterations. Generally, a program may contain the following sorts of outer loops (with respect to this requirement): without recursive dependences (1), with false dependences (2), and with real dependences (3).

The first sort of loops is typical for calculation of explicit steps of algorithms, as well as for steps with implicit dependences in other directions than the outer one. The same applies to the second sort of loops: false dependences appear, for example, in optimized algorithms when numerical flux is calculated only for one face of each computational cell (in every direction) and this value is used later as a corresponding face value of the adjacent cell. Such false dependences can be easily eliminated by introducing flag values indicating whether an outer loop iteration is the first one for the current thread. In this case all necessary cell values must be additionally calculated. The overhead of this technique is low: e.g. if the outer dimension $N = 120$ is split into 8 subdomains, then only every 15'th face value for the corresponding direction must be recalculated.

The case with real dependences between outer loop iterations is the most complex one. This case appears in implicit algorithms and, in particular, in solving the pressure Poisson equation. In the current implementation both algorithm steps (ADI-implicit and Poisson) need to solve tridiagonal linear systems along the outer dimension.

In order to solve a tridiagonal system, the Gauss elimination method is usually used. However, this procedure is intrinsically recursive. The easiest way to overcome recursion is to perform elimination simultaneously from two ends of a tridiagonal system. This method is called twisted factorization. In order to increase the level of parallelization to 4, a hierarchical extension of this technique can be used, when twisted factorizations are applied to both halves of a tridiagonal system [3]. In this case, some fill-in appears, that requires an additional step of the algorithm. This is a two-way parallel partition method. Figure 3 represents twisted factorization after elimination step (left), and two-way method after elimination and backsubstitution steps (center and right, respectively).

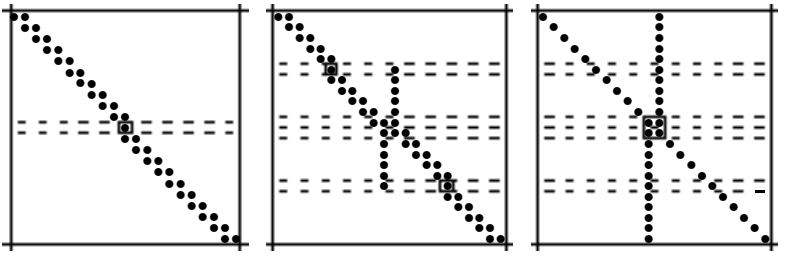


**Fig. 3.** Illustration of the two-way method of parallelization for 2 and 4 threads

Generally, the two-way method can be extended to 8 threads and more. However, this extension greatly complicates the algorithm. In the current approach, splitting of a computational domain by the last (radial) dimension is limited to 4 for recursive algorithms, and additional splitting is performed by another (axial) dimension. For 8-thread parallelization, it is enough to additionally split a domain into 2 parts. For 12 or 16 processor cores, splitting into 3 or 4 parts is used (Fig. 2, right). Further extension to 24 or 32 threads is possible and straightforward, but in this case the parallel efficiency will be limited because of narrowness of subdomains in one-dimensional splitting of the main part of the algorithm (see Fig. 2, left as an illustration of narrow splitting).

Thus, the considered approach uses two techniques: natural parallelization on any number of threads with one-dimensional splitting for simple non-algebraic parts of the algorithm, and special parallelization of tridiagonal linear systems with particular splittings (2, 4, 8, 12, 16 threads) for algebraic parts. In solving the pressure Poisson equation, this approach combines easiness and effectiveness of parallelization with absence of restrictions on grid size and uniformity.

## 5 Results and Analysis

The developed parallelization method was tested and evaluated on several shared memory computer systems. Five of these systems were selected for a comparative analysis:

- cluster node with two 4-core processors Intel Xeon of the Core 2 Quad family (3 GHz, memory 4×DDR2-800);
- 6-core processor Intel Core i7-980X (3.33 GHz, memory 3×DDR3-1333);
- 4-core processor Intel Core i7-920 (2.66 GHz, memory 3×DDR3-1333);
- NuMA-server SGI Altix 350 built on nodes with two Itanium processors in each (1.5 GHz, memory 2×DDR2-533 per node) [6];
- NuMA-server Bull S6030 built on 8-core processors Intel Xeon X7560 as shared-memory nodes (2.27 GHz, memory 4×DDR3-1066 per processor).

Results of measurement are presented on Fig. 4. For this evaluation, a CFD application program with the grid size $128 \times 100 \times 120$ was used.

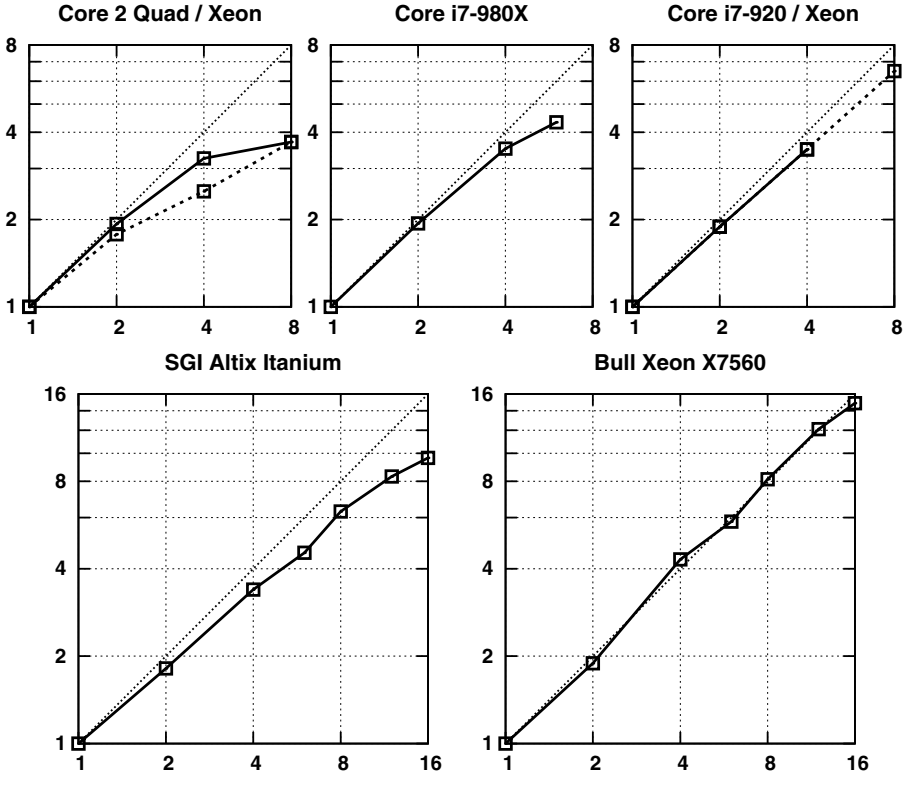



**Fig. 4.** Parallelization speedup for five computer systems

Each system is characterized by particular properties. For example, Core i7-9xx family processor is the most suitable one for the considered problems: it has 4 to 6 fast cores and powerful memory subsystem with high throughput. The latter property is important for memory-bound applications. As a result, this processor demonstrates good speedup and efficiency of parallelization: 3.5 and 87% for 4 threads. However, with 6 threads (Core i7-980X), scalability is not so good because of saturation of the memory subsystem: speedup and efficiency are 4.33 and 72% in this case.

There exist two-processor configurations built on similar processors (Xeon 55xx/56xx families). They belong to the NuMA class because each processor has its own local memory. Due to this, memory throughput increases twice and scales with the number of processor cores. Prediction of the parallel performance for bi-processor Xeon system is shown on Fig. 4 (Core i7-920) by dashed line.

On the contrary, the system with two Xeon processors of the Core 2 Quad family behaves poor because of the old-fashioned bus-based system organization. This organization severely limits memory throughput and, consequently, multi-threaded performance. Solid line on the graph for Core 2 Quad demonstrates degradation of parallel efficiency for 8 threads to only 46%. Dashed line represents speedup when only a corresponding part of the system is allocated to a job (one processor for 4 threads, one dual-core chip of the two-chip processor for 2 threads). This is an additional illustration of limited scalability of this system.

Scalability of Itanium-based SGI Altix non-uniform memory system was expected to be not high for 2 threads because this system is built on bi-processor nodes with limited throughput of their local memories. In fact, it happened to be reasonable: speedup and efficiency are 1.82 and 91% in this case. For 4 and 8 threads, memory subsystem scales linearly that results in good scaling of the parallel performance (by 1.87 for 2 to 4 threads, by 1.86 for 4 to 8 threads). Further increase to 12 and 16 processors doesn't demonstrate so good scalability because additional splitting of a computational domain in axial direction into 3 or 4 parts leads to non-optimal allocation of data and increases data access overhead (due to bi-processor nature of shared-memory nodes). Nevertheless, SGI Altix still demonstrates additional acceleration with 20 and 24 processors.

The best results were obtained for Bull S6030 NuMA-server (Table 1). This system demonstrates superlinear speedup for up to 12 threads. It can be explained by the properties of hierarchical cache and memory subsystems. In particular, large L3-cache (24 MByte) with scalable multibank organization as well as scalable memory controllers distinguish this processor as very suitable for parallelization. Organization of a multiprocessor system with individual memory controllers ensures good parallelization efficiency when more than one processor is used (i.e. for more than 8 threads). Slight degradation of parallel efficiency is observed only for 16 threads and more. Partly this degradation can be explained by non-ideal load balance after splitting of a domain into large number of subdomains (for example, splitting of dimension 120 by 16 gives 7.5 as a subdomain's thickness). Despite this, Bull server demonstrates good parallel efficiency even for 24 threads (86.2%) and reasonable efficiency for 32 threads (71.7%).

**Table 1.** Parallelization results for Bull S6030 multiprocessor server (computational times are presented for one unit of work equal to 26 time steps)

| # threads | 1 | 2 | 4 | 6 | 8 | 12 | 16 | 24 | 32 |
|---|---|---|---|---|---|---|---|---|---|
| total time (sec) | 19.85 | 10.52 | 4.605 | 3.415 | 2.44 | 1.64 | 1.335 | 0.96 | 0.865 |
| speedup | – | 1.89 | 4.31 | 5.81 | 8.14 | 12.10 | 14.87 | 20.68 | 22.95 |
| efficiency | – | 94.3% | 107.8% | 96.9% | 101.7% | 100.9% | 92.9% | 86.2% | 71.7% |
| pressure part (sec) | 3.74 | 1.96 | 0.92 | 0.74 | 0.51 | 0.35 | 0.29 | 0.23 | 0.20 |
| pressure part (% of total) | 18.8% | 18.6% | 20.0% | 21.7% | 20.9% | 21.3% | 21.7% | 24.0% | 23.1% |

Table 1 also presents computational times for pressure/velocity correction procedure (mostly for solving the pressure Poisson equation). It can be seen that proportion of total time spent in this procedure increases with the number of threads due to complexity of parallelization of the tridiagonal solver. Additional increase occurs for 24 and 32 threads because parallelization level for tridiagonal systems is limited to 16 in the current implementation of the algorithm.

Thus, most of the systems above can be efficiently used for parallel computations of this class of numerical problems. In particular, single- and dual-processor computers built on Intel i7-9xx family processors are good and inexpensive candidates for 8-12 thread parallelization, while more expensive multiprocessor NuMA computers with scalable memory subsystems can be considered as target platforms for running 16 threads and more. Very soon, new desktop and small-server processors will appear, such as AMD Bulldozer and Intel Sandy Bridge E. The latter, for example, will have up to 8 processor cores and very powerful memory subsystem with four DDR3 controllers. Forthcoming availability of these systems again justifies simple and efficient OpenMP-based parallelization methods for 12-16 threads.

## 6 Conclusion

In this work we have developed an OpenMP parallelization method for modeling of incompressible flows. The new method is suited for modern multicore processors and multiprocessor shared memory systems of moderate class (with up to 16 processor cores). It employs new efficient direct method for solving Poisson equation and other optimization techniques. The method is adapted to non-uniform memory computers (NuMA) and demonstrates good parallelization efficiency. With the new method, computational speed of a modern inexpensive desktop computer can exceed the speed of a typical computer in 2005 by about 10 times. This method was used for performing massive parametric numerical investigations of hydrodynamic interactions and instabilities in Czochralski model in parallel regimes with up to 16 threads.

**Acknowledgements.** This work was supported by the Russian Foundation for Basic Research (project RFBR-09-08-00230). Access to Altix 350 and Bull S6030 computer systems was granted by Laboratoire de Mécanique, Modélisation et Procédés Propres (M2P2), Marseille, France.

## References

1. Polezhaev, V., Bessonov, O., Nikitin, N., Nikitin, S.: Convective Interaction and Instabilities in GaAs Czochralski Model. J. Crystal Growth 230, 40–47 (2001)
2. Bessonov, O., Polezhaev, V.: Modeling of Three-Dimensional Thermocapillary Flows in Czochralski Method. Transactions of Higher Schools. North Caucasian Region. Natural Sciences. Special Issue. Mathematics and Condensed Matter, 60–67 (2004) (in Russian)
3. Bessonov, O., Brailovskaya, V., Polezhaev, V., Roux, B.: Parallelization of the Solution of 3D Navier-Stokes Equations for Fluid Flow in a Cavity with Moving Covers. In: Malyshkin, V.E. (ed.) PaCT 1995. LNCS, vol. 964, pp. 385–399. Springer, Heidelberg (1995)
4. Bessonov, O., Fougère, D., Roux, B.: Parallel Simulation of 3D Incompressible Flows and Performance Comparison for Several MPP and Cluster Platforms. In: Malyshkin, V.E. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 401–409. Springer, Heidelberg (2001)
5. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering 5(1), 46–55 (1998)
6. Accary, G., Bessonov, O., Fougère, D., Meradji, S., Morvan, D.: Optimized Parallel Approach for 3D Modelling of Forest Fire Behaviour. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 96–102. Springer, Heidelberg (2007)
7. Accary, G., Bessonov, O., Fougère, D., Gavrilov, K., Meradji, S., Morvan, D.: Efficient Parallelization of the Preconditioned Conjugate Gradient Method. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 60–72. Springer, Heidelberg (2009)

# On Quantitative Security Policies*

Pierpaolo Degano, Gian-Luigi Ferrari, and Gianluca Mezzetti

Dipartimento di Informatica, Università di Pisa, Italy
{degano,giangi,mezzetti}@di.unipi.it

**Abstract.** We introduce a formal framework to specify and enforce *quantitative* security policies. The framework consists of: (*i*) a stochastic process calculus to express the measurable space of computations in terms of Continuous Time Markov Chains; (*ii*) a stochastic modal logic (a variant of CSL) to represent the bound constraints on execution speed; (*iii*) two enforcement mechanisms of our quantitative security policies: *potential* and *actual*. The potential enforcement computes the probability of policy violations, thus providing a sort of static evaluation of when the policy is obeyed. This supports the user to accept/discard a component when the probability of the security violation is below/above a suitable chosen threshold. The actual enforcement computes at run-time the deviation of the execution speed from the acceptable rate. This specifies the execution monitor and drives it to abort unsafe executions.

## Introduction

In the last few years a new trend is emerging, that exploits the network for computing in a different manner. Applications are no longer built as monolithic entities, rather they are constructed by plugging together computational facilities and resources offered by (possibly) untrusted providers. Illustrative examples of this approach are the Service Oriented, GRID and CLOUD paradigms. Since applications have little or no control of network facilities, security issues became even more acute. The literature has several proposals that address these problems. They can be roughly divided into dynamic, that monitor executions possibly stopping them when unsecure; and static, that analyse at binding time the published behavioural interfaces to avoid risky executions.

A language based approach supporting the static analysis of security has been developed in [12,11,10,9]. Its main ingredients are: local policies, call-by-contract invocation, type-effect systems, model checking and secure orchestration. However, this approach only takes into account qualitative aspects of behaviour, neglecting quantitative ones, typically the rates at which the different activities are performed. The importance of describing also quantitative aspects of systems is witnessed by several quantitative models and analysis tools that have recently been put forward in the literature. To cite only a few, the stochastic process algebras PEPA[23], the Stochastic $\pi$-calculus [30], EMPA [14], the stochastic model checker PRISM[25].

* This work has been partially supported by IST-FP7-FET open-IP project ASCENS and RAS L.R. 7/2007 project TESLA.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 23–39, 2011.

In this paper we extend the approach of [10] to also deal with quantitative aspects. Our starting point is the abstraction of system behaviour, called *history expression*, that are processes of a suitable process calculus.

We extend history expressions by associating a *rate* with actions, so landing in the world of stochastic process calculi. In this way, we obtain *stochastic history expressions* (HE$\mu$). Our first goal is to give them a quantitative semantics in terms of continuous-time Markov chains (CTMC), so making usable well-known techniques for quantitative analysis [8,26,18]. We use a variation of the stochastic kernels over measurable spaces [15,29] to represent CTMC in the style of [17,16]. To overcome the difficulties with recursion, we restrict stochastic history expressions to a disciplined iteration, namely *binary Kleene star* (for a different approach, see [17]). As a matter of fact, HE$\mu$ turn out to be a stochastic extension of $\mathrm{BPA}^*_\delta$ [22,7].

Our second main contribution is sharpening security policies with quantitative constraints. Roughly, quantitative security policies are safety properties that enforce bounds on the speed at which actions have to be performed. These policies are first class operators inside HE$\mu$, so that security can be taken into account from the very beginning of application development. To express policies we consider $\mathrm{CSL}_S$, a linear subset of CSL [8,4].

Because of the inherent stochasticity of our programming model, policies are to be controlled in two complementary modalities: *potential* or *actual*. The first one applies to the CTMC semantics, hence the check is on the *expected* behaviour — rates in the CTMC associated with the an HE$\mu$ expression $e$ represent the *average* speed of the actions in $e$. Potential analysis then measures the probability of policy violations. This kind of verification can be carried out through a probabilistic model checker, e.g. PRISM [25].

The actual control can only be done dynamically, because in a specific, unlikely computation, the actual speed of an action can greatly deviate from its rate. Security is then enforced during the execution through an *execution monitor* aborting such a unlikely, unsafe computation.

Potential verification enables a user to accept/discard an application when the probability of a security violation is below/above a certain threshold he feels acceptable. Complementary, actual monitoring will stop the unwanted execution, so guaranteeing security.

To clarify our formal development, we introduce below a simple example. A more detailed one will be illustrated in Section 4. We want to analyze a system, the behavior of which is specified by the following process. The system starts a race between actions $a$ and $c$. In the case $a$ is the first to complete, $b$ is performed and then the whole process restarts. Otherwise, if $c$ wins the race, the process restarts right after $c$ completion. We model the expected execution speed by associating to each action a rate, used then as the parameter of an exponentially distributed random variable. Section 1 reviews the basic notions about random variables and probability theory. For simplicity, suppose that here the action $a$ has rate 0.5, while the actions $b$ and $c$ have both rates 1.

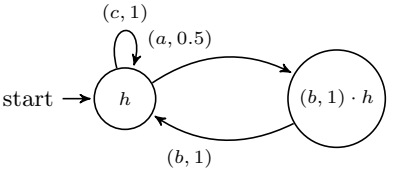



**Fig. 1.** CTMC associated with $h$

We model the system above through the following HE$\mu$ expression $h = (((a, 0.5) \cdot (b, 1)) + (c, 1))^{*} \delta$. The operator $^{*}$ is the binary Kleene star, that expresses the iteration of the process (the $\delta$ is the deadlock process preventing the iteration to terminate). As said, the long term behavior of HE$\mu$ expressions is conveniently specified by a CTMC. In our case, we give a graphical representation of the semantics of $h$ in Figure 1. The syntax and the semantics of the stochastic history expressions are formally defined in Section 2.

Assume now that the system has to respect a quantitative actual policy $\phi$ saying that "action $a$ must never last more than 1 second". This policy is to be reflected into a potential requirement, expressing that, in the long-run, the system will violate $\phi$ with low probability. A suitable $\text{CSL}_S$ formula that represents this potential quantitative policy is $\psi = \mathcal{C}_{\leq 1\%}(\phi)$. We omit here the details and only read $\psi$ as: the computations violating $\phi$ are less than 1% of the total. Section 3 presents the way we specify and enforce policies.

We now verify whether the expression $h$ respects the potential policy $\psi$ or not. To this purpose, we compute the vector of the steady state distribution of the CTMC associated with $h$. Each entry of the vector expresses the portion of time spent in each part of the computation. The steady distribution of the CTMC in Figure 1 is $[0.\bar{6}, 0.\bar{3}]$. The first entry is related with the part where $a$ and $c$ are racing, the second one with the part when $b$ is executing. By standard reasoning on the properties of exponential random variables, the probability that $a$ lasts longer than 1 second is $p = 0.36$, the probability that the action $a$ wins the race is $q = 0.\bar{3}$. Hence, we obtain that $\psi$ is violated because the probability that $\phi$ is violated is about 8%. Indeed, multiplying $0.\bar{6}$, the first entry of the vector (when action $a$ is executing), by $q$ (the probability that $a$ is the one that completes) and by $p$ (the probability that the duration of $a$ violates $\phi$) we get about 0.08.

This analysis suggests to deploy the system equipped with a monitoring mechanism that abort an execution when it is about to violate $\phi$. More details are in Section 3.

## 1 Preliminaries

We review the main notions and notations about measure theory and we refer the reader to [3,2] for more details.

Given the *support set* $M \neq \emptyset$, a *$\sigma$-algebra* $\Sigma$ over $M$ is a set of subsets of $M$, the *measurable sets*, containing $\emptyset$ and closed under complement and countable

union. The structure $\mathcal{M} = (M, \Sigma)$ is a *measurable space* and a *measure* over it is a function $\kappa : \Sigma \to \mathbb{R}^+ \cup \{\infty\}$ such that:

1. $\kappa(\emptyset) = 0$
2. Given a countable collection $\{N_i\}_{i \in I}$ of pairwise disjoint sets in $\Sigma$ then: $\kappa(\cup_{i \in I} N_i) = \sum_{i \in I} \kappa(N_i)$ ($\sigma$-*additivity*)

The class of measures on a measurable space $\mathcal{M}$ will be denoted by $\Delta(M, \Sigma)$ or $\Delta(\mathcal{M})$ when the support set and the $\sigma$-algebra are clear from the context.

Given a class of sets $G$, called *generator*, $\sigma(G)$ is the minimal $\sigma$-algebra containing $G$. If $G$ contains all pairwise disjoint sets then $G$ is a *base* of $\sigma(G)$. Note that $\sigma(G)$ always exists since $\wp(G)$ contains $G$ and the intersection of an arbitrary collection of $\sigma$-algebras is a $\sigma$-algebra.

Given two measurable spaces $(M_1, \Sigma_1), (M_2, \Sigma_2)$ a function $f : M_1 \to M_2$ is *measurable* iff $\forall A \in \Sigma_2 . f^{-1}(A) \in \Sigma_1$. The class $\|M_1 \to M_2\|$ contains the measurable functions between $(M_1, \Sigma_1)$ and $(M_2, \Sigma_2)$ omitting $\Sigma_1, \Sigma_2$ when unambiguous. A measurable function is structure-preserving.

Hereafter, whenever using a measurable space of measures $\kappa \in \Delta(M, \Sigma)$, we will consider the $\sigma$-algebra generated by the sets $\{\kappa \in \Delta(M, \Sigma) \mid \kappa(S) \geq r\}$ for arbitrary $S \in \Sigma, r > 0$, .

Given an arbitrary set $\Omega \neq \emptyset$, the *sample space*, a $\sigma$-algebra on it and a measure $\mathbb{P}$ s.t. $\mathbb{P}(\Omega) = 1$, we can build a probability space $(\Omega, \Pi, \mathbb{P})$ so interpreting standard probability theory in the measure theoretic context. For instance $\mathbb{P}(A)$ is the "probability of the events in $A$" with $A$ measurable, i.e. $A \in \Pi$.

The random variables of probability theory can be defined and generalized in measure theory. A random variable $X$ is a measurable function between $(\Omega, \Pi)$ and the measurable space formed by intervals in $\mathbb{R}$. The measure $\mathbb{P}$ governs the probability of $X^{-1}([a, b])$. Given a comparison symbol $\lhd \in \{\leq, \geq, <, >\}$, we abbreviate $\mathbb{P}(X^{-1}(\{x \mid x \lhd z\}))$ with $\mathbb{P}(X \lhd z)$.

A random variable of parameter $\lambda$ has an exponential distribution if

$$\mathbb{P}(X \leq t) = \begin{cases} 1 - e^{-\lambda t} & \text{if } t \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Only exponentially distributed random variables enjoy the *memoryless property* $\mathbb{P}(X > s + t \mid X > t) = \mathbb{P}(X > s)$ for all $s, t \geq 0$; they have *mean* $\frac{1}{\lambda}$; if $\{X_i\}_{i \in I}$ is a finite set of such variables with parameter $\lambda_i$ then $M = \min\{X_i\}_{i \in I}$ is an exponential random variable with parameter $\sum_{i \in I} \lambda_i$.

We now introduce Continuous-Time Markov chains on a countable state space. We refer to [28,24,3,8] for details.

**Definition 1.1.** *Let $S$ be a denumerable set, a* Continuous-Time Stochastic Process *is an indexed (by $t \in \mathbb{R}^+$) family of random variables $X(t) : \Omega \to S$ with $(\Omega, B, \mathbb{P})$ probability space.*

The value of the variable $X_t$ can be interpreted as the state of the process at time $t$. A continuous-time system evolves performing transitions between states. We limit ourself to specific class of processes with the *Markov* property.

**Definition 1.2.** *A* Continuous-Time Markov Chain *(CTMC) is a stochastic process $X(t)$ with $t \geq 0$ such that for any $s, t \geq 0$ and $i, j, x_u \in S$ the* Markov property *holds :*

$$\mathbb{P}\left(X(t+s) = j \mid X(s) = i, X(u) = x_u, 0 \leq u < s\right) = \mathbb{P}\left(X(t+s) = j \mid X(s) = i\right)$$

*The CTMC is said to have* homogeneous *transition probabilities if:*

$$\mathbb{P}(X(t+s) = i \mid X(s) = j) = \mathbb{P}(X(t) = i \mid X(0) = j)$$

Hereafter we will only use homogeneous CTMC.

Because of the Markov property, we denote with a random variable $L_j$, depending only on the current state $j$, the *sojourn time*: the amount of time spent in $j$ before performing a new transition. Also we have that:

$$\mathbb{P}\left(L_i > s + v \mid L_i > s\right) = \mathbb{P}\left(L_i > v\right) \qquad \text{memoryless property of } L_i$$

Hence, it turns out that $L_i$ must be *exponentially* distributed. When the process leaves state $i$, it can reach another state with a certain probability. This probability does not depend on the time spent in $i$. Hence we will indicate with $p_{ij}$ the probability that the process reach state $j$ from state $i$.

As a consequence, a CTMC is completely characterized by the parameters $\lambda_i$ of the exponentially distributed random variables $L_i$ and by $p_{ij}$. A well-know representation of a CTMC is the *rate-matrix* $R = [r_{ij}]$ with $r_{ij} = p_{ij}\lambda_i$.

This representation suggests another interpretation of the evolution of a CTMC. Since the minimum of a set $\{C_i\}_{i \in I}$ of exponentially distributed random variables with rate $c_i$ is again an exponentially distributed random variable with rate $\sum_{i \in I} c_i$, we can interpret $L_i$ as the minimum $\min\{R_{ij}\}$ of a set of random variables with rate $r_{ij}$. Hence, a CTMC models a process that, while entering in state $i$, enables a set of action, whose durations is modelled by the random variables $\{R_{ij}\}_{j \in S}$. These actions are competing (*racing*) for completion. If the action with duration $R_{ij}$ is the faster, then the next state will be $j$.

Another common and useful characterization of CTMC is through the infinitesimal generator matrix $Q$, given in terms of the rate matrix:

**Definition 1.3.** *Let $D = [d_{ij}]$ be a matrix with $d_{ij} = 0$ if $i \neq j$ and $d_{ii} = \sum_{j \in S} r_{ij}$ otherwise. Then the* infinitesimal generator matrix *is $Q = R - D$.*

All previous definitions smoothly extend considering a function $L : S \times S \to A$ labelling transitions between states.

Let $p_{ij}(t) = \mathbb{P}(X(t) = j \mid X(0) = i)$ and $P(t) = [p_{ij}(t)]$ its matrix representation, a *steady distribution* $\pi$ is a vector such that $\pi = \pi P(t)$ for all $t \geq 0$. The meaning is that if we use $\pi$ as distribution of $X(0)$ then this will remain the same for all $t > 0$:

$$\mathbb{P}(X(t) = j) = \sum_{i \in S} \mathbb{P}(X(t) = j \mid X(0) = i) \cdot \mathbb{P}(X(0) = i) = \sum_{i \in S} p_{ij}(t)\pi_i = [\pi P(t)]_j = \pi_j$$

It is well know that the infinitesimal generator matrix plays almost the same role (in Continuous-Time) of the probability matrix of a Discrete-Time Markov Chain finding a steady distribution:

***Fact.*** *Given a CTMC let $Q$ be its infinitesimal generator matrix, then $\pi$ is its steady distribution iff $\pi Q = 0$ and $\sum_{i \in S} \pi_i = 1$*

Such $\pi$ always exists for a finite CTMC and it is unique if the CTMC is irreducible (if every state can be reached by a sequence of transitions from all other states). A steady distribution is important because a CTMC will reach it "on the long run" as shown by the following theorem:

***Fact.*** *If a CTMC has a steady distribution $\pi$, then $\forall i : \lim_{t \to \infty} p_{ij}(t) = \pi_j$*

The notion of steady distribution is very important to analyse reliability and performance of a system modelled as a CTMC. In fact, $\pi_j$ represents the proportion of time spent in state $j$ on the long run. This information can be used to infer the typical behaviour of the system.

## 2 Stochastic History Expressions

A language-based framework for managing security issues in a distributed contexts has been proposed in [12,11,10,9]. The starting point of these works is a functional programming language supporting remote service invocation and the enforcing of security policies. The execution of a distributed application comprise local and remote computations. Security relevant events generated during the executions of the application are collected in sequences, called *histories*. Security policies express constraints over these histories. Enforcing security can be done statically and dynamically. The dynamic mechanism uses a runtime monitor that blocks executions about to violate a security requirement. The static one uses a formalism, called *history expressions*, to represent all the histories that can be generated. These are then model checked to verify whether the constraints will be always satisfied. Their approach is qualitative only, here we present a first step towards a quantitative extension.

We first extend histories into *timed histories*. A timed history is a possibly empty sequence $((a, t_a), (b, t_b), \dots)$ of occurred events, with $t_x$ duration of event $x$. Also security policies are extended to express timing constraints on timed histories. The values $t_x$ are unpredictable on a single run of a program but we assume these duration to be exponentially distributed.

Under this assumption, we extend history expressions and obtain *stochastic history expressions* (HE$\mu$). These express in a finite way potentially infinite timed histories and enable us to model check quantitative policies using well-known techniques. Indeed, the semantics of a HE$\mu$ process is given in terms of a CTMC, that implicitly describes both the timed histories and the long run behaviour of a program. It is convenient to use a functional representation of CTMC called *Markov kernel* (see Definition 2.4).

### 2.1 Syntax

The building blocks of HE$\mu$ are stochastic events. Given an alphabet of event names $\mathbb{A}$, a stochastic event is a pair $(a, \alpha) \in \mathbb{A} \times \mathbb{Q}^+$, where $\alpha$ is the rate of $a$, i.e. the parameter of the exponential random variable modelling its duration.

**Definition 2.1.** *A stochastic history expressions (*HE$\mu$ *)* $h \in \mathbb{H}$ *is a term generated by the following grammar:*

$$
\begin{array}{lllllll}
h_1, h_2 & ::= (a, \alpha) & \textit{(stochastic event)} & | & \delta & \textit{(deadlock)} & | \\
 & \psi[e_1] & \textit{(policy framing)} & | & h_1 \cdot h_2 & \textit{(sequentialization)} & | \\
 & h_1 + h_2 & \textit{(stochastic choice)} & | & (h_1 \,^* h_2) & \textit{(binary Kleene star)} &
\end{array}
$$

A stochastic event $(a, \alpha)$ performs action $a$ and then successfully terminates. *Deadlock* $\delta$ is a non terminated process that cannot perform any action. The *stochastic choice* operator + simultaneously enables two or more actions of processes $h_1, h_2$. We consider the enabled actions as competing: this means that the system is in a *race condition* and it hangs waiting for the fastest action to occur, while discarding the slower ones. We also consider a disciplined form of iteration: the binary Kleene star. It takes two processes and let them race. If the left process wins then it executes and the race starts again, otherwise the right executes and the iteration is over. We can express infinite behaviour as the *no-exit iteration* $(h_1 \,^* \delta)$ [13] that describes a process continuously doing $h_1$. For an overview about expressiveness of iteration, recursion, replication and a comparison between unary and binary star in classical process algebras see [21,6]. The sequentialization operator is often present in stochastic process algebras in its restricted variant of action *prefix*. However in [10] it is crucial to define the type-and-effect system that associates programs to history expressions. To manage sequentialization we will need the concept of terminated process, indicated with $\checkmark$, that cannot fire any action, still being different from $\delta$. Indeed intuitively $\checkmark \cdot a = a$ while we will make sure that $\delta \cdot h = \delta$, see [1,5]. We remark that we will deal with $\checkmark$ into the semantic definitions and not in the syntax. In this we follow [1] where termination is treated as a meta-predicate over processes. Similarly to history expressions we attach policies $\psi$ to expressions through the framing construct. We will define formally quantitative policies in Section 3.

We choose to stick on exponential distribution because the resulting mathematical theory enjoys elegant properties, e.g. the way we use to break the race condition. Other distributions can also be accommodated in our framework without much effort, especially because we neglect here an explicit parallel operator.

Note that the stochastic choice operator enables us to cast pure probabilistic branching in a stochastic setting. This behaviour can be simulated using stochastic choice in combination with high rate events s.t. the time consumed for their completion is negligible in the analysis, while providing the intended probabilistic behaviour.

In the literature there are many stochastic process algebras: PEPA [23], EMPA [14], stochastic $\pi$-calculus [30] and the stochastic version of CCS in [16], just to cite a few. Our HE$\mu$ algebra differs from these mainly because we offer full sequentialization, quantitative policy framing and the binary Kleene star. As a matter of fact, HE$\mu$ with no policy framing turns out to be to be a stochastic extension of $\text{BPA}^*_\delta$ [22,7].

### 2.2 Structural Equivalence

We first define a structural congruence over the set of processes $\mathbb{H}$.

**Definition 2.2.** *We define the relation $\equiv$ as the smallest relation over $\mathbb{H}$ such that:*

- *It is an equivalence relation and a congruence with respect to $\cdot, +, \psi, ^*$.*
- *It respects the following laws:*

$$\delta \cdot h_1 \equiv \delta \qquad h_1 + \delta \equiv h_1 \qquad h_1 + h_2 \equiv h_2 + h_1$$
$$h_1 + (h_2 + h_3) \equiv (h_1 + h_2) + h_3 \qquad h_1 \cdot (h_2 \cdot h_3) \equiv (h_1 \cdot h_2) \cdot h_3$$
$$(h_1 \cdot h_3) + (h_2 \cdot h_3) \equiv (h_1 + h_2) \cdot h_3 \qquad h_1 \,^* (h_2 \cdot h_3) \equiv (h_1 \,^* h_2) \cdot h_3$$

Note that the law $h_1 \cdot (h_2 + h_3) \equiv (h_1 \cdot h_2) + (h_1 \cdot h_3)$ is missing. Indeed $(a, \alpha) \cdot ((b, \beta) + (c, \gamma)) \not\equiv (a, \alpha) \cdot (b, \beta) + (a, \alpha) \cdot (c, \gamma)$ because $b$ and $c$ are in a race condition within the left process.

We define now some quotient spaces with respect to the structural equivalence that will be used in the semantics:

**Definition 2.3.** *We define $\mathbb{H}^{\equiv}$ to be the set of $\equiv$-equivalence classes of $\mathbb{H}$ and $[h]^{\equiv}$ the equivalence class of $h \in \mathbb{H}$. Given the minimal $\sigma$-algebra $\Xi^{\equiv}$ generated by $\mathbb{H}^{\equiv}$ (i.e. $\Xi^{\equiv} = \sigma(\mathbb{H}^{\equiv})$, the measurable space $\mathcal{H}^{\equiv}$ is $\mathcal{H}^{\equiv} = (\mathbb{H}, \Xi^{\equiv})$. Finally, $\mathbb{H}_{\checkmark}$ indicates $\mathbb{H} \cup \{\checkmark\}$ and $\mathcal{H}^{\equiv}_{\checkmark} = (\mathbb{H}_{\checkmark}, \Xi^{\equiv}_{\checkmark})$ with $\Xi^{\equiv}_{\checkmark}$ the $\sigma$-algebra generated by $\mathbb{H}^{\equiv} \cup \{\checkmark\}$.*

### 2.3 Semantics

We give semantics of HE$\mu$ following the approach of [16,17]. We start with a slight variant of the *Markov Kernel* to accommodate termination and iteration. As a matter of fact, Markov Kernel is a labelled version of *Stochastic Kernel* introduced in [15,29].

**Definition 2.4.** *Given a measurable space $\mathcal{M} = (M, \Sigma)$ and the denumerable set $\mathbb{A}$ of event names, let $\Sigma' = \sigma(\Sigma \cup \{\checkmark\})$ be the smallest $\sigma$-algebra over $M' = M \cup \{\checkmark\}$ containing the sets in $\Sigma$ and the singleton $\{\checkmark\}$. A* Markov kernel *is a triple $(\mathbb{A}, \mathcal{M}, \theta)$ where $\theta : \mathbb{A} \to \|M \to \Delta(M', \Sigma')\|$ is its* transition function. *A* Markov Process *is a quadruple $(\mathbb{A}, \mathcal{M}, \theta, m)$ where $m \in M$ is the initial state.*

To give a compact definition of semantics to HE$\mu$, it is convenient to introduce some auxiliary notations. Recall that it suffices to define a measure on $G$ to obtain its extension on $\sigma(G)$, which follows by $\sigma$-additivity.

- The *r-Dirac measure* on $N$ is defined: $\delta^r_N(N') := \begin{cases} r & \text{if } N' = N \\ 0 & \text{otherwise} \end{cases} \quad \forall N' \in G$
- The *null measure* is defined: $\omega(N') := 0 \quad \forall N' \in G$
- Given an alphabet $\mathbb{A}$ we define the function $\omega^{\mathbb{A}} : \mathbb{A} \to \Delta(\mathcal{H}^{\equiv}_{\checkmark})$ such that $\omega^{\mathbb{A}}(x) = \omega$, with $\omega$ null measure on $\mathcal{H}^{\equiv}_{\checkmark}$.

– Given $a \in \mathbb{A}$ the function $[^a{}_\kappa] : \mathbb{A} \to \Delta(\mathcal{H}^{\equiv}_{\checkmark})$ is such that

$$[^a{}_\kappa](x) = \text{if } x = a \text{ then } \kappa \text{ else } \omega$$

In the following we will use this operator instantiated in $[^a{}_{\delta^\alpha_\checkmark}]$, with $\delta^\alpha_\checkmark$ $\alpha$-Dirac delta measure on $\checkmark$.

– Given $h \in \mathbb{H}$, any function in the class of $h$-operator $\odot_h : \Delta(\mathcal{H}^{\equiv}_{\checkmark})^{\mathbb{A}} \to \Delta(\mathcal{H}^{\equiv}_{\checkmark})^{\mathbb{A}}$ is defined as follows:

$$[\odot_h \mu](a)(H) = \sum_{k \in H} \begin{cases} \mu(a)([l]^{\equiv}) & \text{if } \exists l \in \mathbb{H}. k \equiv l \cdot h \\ \mu(a)(\checkmark) & \text{if } k \equiv h, k \not\equiv \delta \\ 0 & \text{otherwise} \end{cases}$$

– The binary operator $\oplus : \Delta(\mathcal{H}^{\equiv}_{\checkmark})^{\mathbb{A}} \times \Delta(\mathcal{H}^{\equiv}_{\checkmark})^{\mathbb{A}} \to \Delta(\mathcal{H}^{\equiv}_{\checkmark})^{\mathbb{A}}$ is defined as follows:

$$(\mu \oplus \mu')(a)(H) = \mu(a)(H) + \mu'(a)(H)$$

The operators are well-defined and enjoy the following properties.

**Lemma 2.1 (Properties of the operators)**

1. $\odot_h \omega = \omega$
2. $\mu \oplus \mu' = \mu' \oplus \mu$
3. $(\mu \oplus \mu') \oplus \mu'' = \mu \oplus (\mu' \oplus \mu'')$
4. $\mu \oplus \omega^{\mathbb{A}} = \mu$
5. $\odot_h \mu = \odot_{h'} \mu$ *if* $h \equiv h'$
6. $\odot_h(\mu \oplus \mu') = (\odot_h \mu) \oplus (\odot_h \mu')$
7. $\odot_{(h_1 \cdot h_2)} \mu = \odot_{h_2} \odot_{h_1} \mu$

In our semantic context, as in [16], Structural Operational Semantics (SOS) rules are not used to give a pointwise semantics ($P \to Q$), rather they define a function that map processes to rate distributions (the *Markov kernel*).

We now introduce SOS rules to map HE$\mu$ processes to functions in $\Delta(\mathcal{H}^{\equiv}_{\checkmark})^{\mathbb{A}}$. Indeed we are defining a relation $\rightsquigarrow \subseteq \mathbb{H} \times [\mathbb{A} \to \Delta(\mathcal{H}^{\equiv}_{\checkmark})]$. If $(h, \mu) \in \rightsquigarrow$, the intended meaning of $\mu(a)(K)$ is the total apparent rate (sum of all rates) of an $a$-transition from $h$ to a state in $K$.

**Definition 2.5.** *The relation* $\rightsquigarrow \subseteq \mathbb{H} \times [\mathbb{A} \to \Delta(\mathcal{H}^{\equiv}_{\checkmark})]$ *is the smallest relation satisfying the following rules:*

$$(ddk)\frac{}{\delta \rightsquigarrow \omega^{\mathbb{A}}} \qquad (act)\frac{}{(a,\alpha) \rightsquigarrow [^a{}_{\delta^\alpha_\checkmark}]} \qquad (cho)\frac{h_1 \rightsquigarrow \mu_1 \quad h_2 \rightsquigarrow \mu_2}{h_1 + h_2 \rightsquigarrow \mu_1 \oplus \mu_2}$$

$$(seq)\frac{h_1 \rightsquigarrow \mu}{h_1 \cdot h_2 \rightsquigarrow \odot_{h_2}(\mu)} \qquad (cnt)\frac{h \rightsquigarrow \mu}{\psi[h] \rightsquigarrow \mu} \qquad (star)\frac{h_1 \rightsquigarrow \mu_1 \quad h_2 \rightsquigarrow \mu_2}{h_1 \,^* h_2 \rightsquigarrow [\odot_{(h_1 {}^* h_2)} \mu_1] \oplus \mu_2}$$

We briefly comment on the above rules. The semantics of $\delta$ is a function that associates the null measure with any action in $\mathbb{A}$; therefore $\delta$ only has transition with rate 0, i.e. it can fire no transition. Instead, $(a, \alpha)$ has an $a$-transition towards $\checkmark$ with rate $\alpha$, while the others have rate 0. Sequentialization of $h_1, h_2$ builds the associated function following a sort of continuation semantics. The quantitative policy $\psi$ is neglected at this semantic level, yet maintaining at a

**Table 1.** SOS derivation of $((\mathrm{a},2)+(\mathrm{b},1.5))\cdot(\mathrm{c},3)$

$$\text{(seq)}\ \frac{\text{(cho)}\ \dfrac{\text{(act)}\ \dfrac{\mu_1 = [{}^{\mathrm{a}}{}_{\delta^{2}_{\surd}}]}{(\mathrm{a},2)\rightsquigarrow\mu_1} \qquad \text{(act)}\ \dfrac{\mu_2 = [{}^{\mathrm{b}}{}_{\delta^{1.5}_{\surd}}]}{(\mathrm{b},1.5)\rightsquigarrow\mu_2}}{((\mathrm{a},2)+(\mathrm{b},1.5))\rightsquigarrow\mu_1\oplus\mu_2}}{((\mathrm{a},2)+(\mathrm{b},1.5))\cdot(\mathrm{c},3)\rightsquigarrow\odot_{(\mathrm{c},3)}[\mu_1\oplus\mu_2]}$$

syntactic level, so paving the way to subsequent security check. Our semantics of the choice operator diverges from known approaches (e.g. multi transition systems in PEPA [23], EMPA [14] and stochastic $\pi$-calculus [30]) because the non-determinism of SOS rules is substituted by a weighted functional approach, where single possibilities are rated and encoded in a function. This functional approach directly associates the correct rate with racing actions, while others need an additional normalization phase. For instance, an external observer looking at the racing process $(a,\alpha)+(a,\alpha)$ would see action $a$ occurring at rate $2\alpha$, while a non-normalized transition system associates with $a$ the rate $\alpha$. Our SOS correctly associates to this process the function $\mu = [{}^{a}{}_{\delta^{\alpha}_{\surd}}]\oplus[{}^{a}{}_{\delta^{\alpha}_{\surd}}]$ such that $\mu(a)(\surd)=2\alpha$. Back to the operational rules, the semantics of the Kleene star is a composition of $\odot$ and $\oplus$.

The following properties will be useful to build a Markov kernel for HE$\mu$.

**Theorem 2.1.** *For any $h\in\mathbb{H}$ there exists a unique $\mu\in\Delta(\mathcal{H}^{\equiv}_{\surd})$ such that $P\rightsquigarrow\mu$.*

*Example 1.* Taken $h=((\mathrm{a},2)+(\mathrm{b},1.5))\cdot(\mathrm{c},3)$, Table 1 shows the derivation of $h\rightsquigarrow\odot_{(\mathrm{c},3)}[\mu_1\oplus\mu_2]$, that indeed is a function. Then, we use the classical tabular representation of functions in Table 2 to represent the meaning of $h$. We show entries only for non-zero values or non-null measures for singleton set of process.

We states that our semantics is correct with respect to structural equivalence.

**Theorem 2.2.** *If $h\equiv h'$ then $h\rightsquigarrow\mu$ and $h'\rightsquigarrow\mu$.*

Now we use the construction given in Definition 2.4 to eventually build the Markov Kernel for HE$\mu$.

**Theorem 2.3.** *Let $h\in\mathbb{H}, a\in\mathbb{A}, H\in\Xi^{\equiv}, H\in\Xi^{\equiv}, \rho$ be a function such that $\rho(a)(h)(H)=\xi(h)(a)(H)$, where $\xi(h)=\mu$ whenever $h\rightsquigarrow\mu$, and $\mathbb{H}^{\equiv},\Xi^{\equiv}$ are as in Def. 2.3. Then $(\mathbb{A},\mathcal{H}^{\equiv},\rho)$ is the* Markov kernel *associated with the HE$\mu$.*

**Table 2.** Tabular representation of semantics

$\mu_1 =$

| | $[\surd]^{\equiv}$ | 2 |
|---|---|---|
| $a$ | ⋮ | ⋮ |
| ⋮ | ⋮ | |

$\mu_2 =$

| | $[\surd]^{\equiv}$ | 1.5 |
|---|---|---|
| $b$ | ⋮ | ⋮ |
| ⋮ | ⋮ | |

$\mu_1\oplus\mu_2 =$

| | $[\surd]^{\equiv}$ | 2 |
|---|---|---|
| $a$ | ⋮ | ⋮ |
| $[\surd]^{\equiv}$ | ⋮ | ⋮ |
| $b$ | ⋮ | ⋮ |
| ⋮ | ⋮ | |

$\odot_{(c,3)}\,[\mu_1\oplus\mu_2] =$

| | $[(c,3)]^{\equiv}$ | 2 |
|---|---|---|
| $a$ | ⋮ | ⋮ |
| $[(c,3)]^{\equiv}$ | ⋮ | ⋮ |
| $b$ | ⋮ | ⋮ |
| ⋮ | ⋮ | |

Eventually we associate a Markov Process with a HE$\mu$ $h$ as follows:

**Definition 2.6.** *$P[\![h]\!]$ is the Markov Process $(\mathbb{A}, \mathcal{H}^{\equiv}, \xi, h)$.*

### 2.4 Rate Bisimulation of HE$\mu$

The notion of behavioural equivalence is of primary importance because it highlights what really matters focusing objects at the right distance, and allows one to substitute equivalent processes preserving the overall behaviour. Additionally, some model checking techniques exploits behavioural equivalence to reduce the state space dimension and this is particularly important for our purpose verifications.

Structural equivalence is too weak. For instance consider the expressions $(a, 2\alpha)$ and $(a, \alpha) + (a, \alpha)$. They represents two process doing action $a$ with the same apparent rate. Hence from an external point of view their behaviour is identical, but clearly $(a, 2\alpha) \not\equiv (a, \alpha) + (a, \alpha)$.

In this work we will use a specific type of bisimulation, globally recognized as the finest equivalence notion [31]. Rate bisimulation used here appears in [16] and generalise *rate aware bisimulation* [19] and probabilistic bisimulation [27].

**Definition 2.7 (Rate bisimulation of HE$\mu$).** *A* rate bisimulation *is an equivalence relation $\mathfrak{R} \subseteq \mathbb{H}^{\equiv} \times \mathbb{H}^{\equiv}$ such that if $(h_1, h_2) \in \mathfrak{R}$ then for all $a \in \mathbb{A}$ and for all measurable subset $H$ that are $\mathfrak{R}$-closed in $\Xi^{\equiv}_{\surd}$:*

$$\rho(a)(h_1)(H) = \rho(a)(h_2)(H)$$

*Two histories $h, h'$ are* rate bisimilar *($h \sim h'$) iff there exist a rate bisimulation $\mathfrak{R}$ such that $(h, h') \in \mathfrak{R}$, thus $\sim$ (*bisimilarity*) is the union of all rate-bisimulations.*

As expected, rate bisimilarity is compatible with structural congruence and with the SOS semantics; also it is preserved under all the operator of HE$\mu$, namely it is a congruence.

**Theorem 2.4**

- *If $h \rightsquigarrow \mu$ and $h' \rightsquigarrow \mu$ then $h \sim h'$.*
- *If $h \equiv h'$ then $h \sim h'$.*
- *The relation $\sim$ is a congruence.*

## 3 Stochastic Security Policies

In this section we will introduce our notion of quantitative securities policy as constraints on the timed behaviour of processes along with two complementary ways to check them. Our first manner is *potential*: the check is made on the CTMC associated with a HE$\mu$ process $h$. This is because, if $h \rightsquigarrow \mu$, the function $\mu$ records the rates of the actions in $h$, and because these rates express the probability that the actions have to fire within a certain time interval.

Since probabilities are computed on the long run, in principle we would like to perform a security check on an always running system. The infinite behaviour of such system is indeed represented by the long run behaviour of the associated CTMC. Then we assume the semantics of the process $h$ under analysis be an irreducible CTMC with a unique steady state. The potential check of security policies is therefore performed on steady states.

However, the duration of actions in an unlikely execution may greatly differ from the speed expresses by their rates. An *actual* check is therefore performed on the execution history $(a, t_a)(b, t_b)\ldots$ by a monitor, which can abort the execution, when about to violate the security policy in hand. For this we assume that timed histories encompass all security relevant events. We shall formalise the above intuition below.

Below we assume as given a CTMC $\mathbf{O}$. Recall that we let $s_i \in S$ be the set of states of $\mathbf{O}$, $R = [r_{ij}]$ be its rate matrix, $Q = [q_{ij}]$ be its infinitesimal generator matrix, $\pi$ be its steady state vector and $D = [d_{ij}]$ be the matrix in Definition 1.3, where, for notational convenience we let $D(i) = d_{ii}$. In addition $I = [a, b]$ is an interval in $\mathbb{R}^+$ , when $\inf I = a \leq b = \sup I$, with possibly $b = \infty$.

### 3.1 Abstracting Executions

A CTMC implicitly represents a set of timed histories associated with the *paths* we define below.

**Definition 3.1 (Paths)** *A* path $\sigma \in \mathbf{Path}$ *over* $\mathbf{O}$ *is an infinite sequence* $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_i} s_{i+1} \ldots$ *with* $\forall i \in \mathbb{N}, s_i \in S$ *and* $t_i \in \mathbb{R}^+$ *such that* $r_{i,i+1} > 0$. *Given a labelling function* $L : S \times S \to \mathbb{A}$, *the path* $\sigma$ *is associated with the timed history* $(a_0, t_0), (a_1, t_1), \ldots$ *with* $a_i = L(s_i, s_{i+1})$. *We write* $\sigma[i]$ *for* $s_i$, $\delta(\sigma, i)$ *for* $t_i$ *and* $\sigma@t$ *for the state* $\sigma$ *at time* $t$.

We construct the following $\sigma$-algebra over paths in order to measure the probability of sets of timed histories.

**Definition 3.2** *Let* $p = (s_0, I_0, \ldots, I_{n-1}, s_n) \in P_I$ *be a sequence of "intervals of paths", i.e. states and intervals, and let* $C(p)$ *be the* cylinder set*: consisting of all paths* $\sigma \in \mathbf{Path}$ *such that* $\forall i \leq n.\sigma[i] = s_i$ *and* $\forall i < n.\delta(\sigma, i) \in I_i$. *Final let* $\Sigma_{\mathbf{Path}}$ *be the* $\sigma$*-algebra generated by the base of cylinder sets* $\{C(p)\}_{p \in P_I}$.

Assume as given a cylinder set $C(p)$ and let $\tau$ be the initial probability distribution over states of the CTMC $\mathbf{O}$. We measure the probability of all paths $\sigma \in C(s_0, I_0, \ldots, I_{n-1}, s_n)$ assuming to be in state $s_0$ with probability $\tau(s_0)$. The sojourn time in state $s_i$ is an exponentially distributed random variable with parameter $D(s_k) = \sum_{k \in S} r_{ik}$. The probability of leaving $s_k$ in the interval $I_k$ is

$$\int_I D(s_k) \cdot e^{-D(s_k)\cdot t}\, dt = e^{-D(s_k)\cdot \inf(I_k)} - e^{-D(s_k)\cdot \sup(b_k)} \quad \text{with } e^{-D(s_k)\cdot\infty} = 0$$

and the probability of choosing as next state $s_{k+1}$ is $p_{k,k+1} = \frac{r_{k,k+1}}{D(s_k)}$.

Finally, the probability to follow a path in $C(s_0, I_0, \ldots, I_{k-1}, s_k)$ is inductively defined on the length of cylinder as:

$$\mathsf{P}_\tau\left(C\left(s_0\right)\right) = \tau(s_0)$$
$$\mathsf{P}_\tau\left(C\left(\ldots, s_k, I_k, s_{k+1}\right)\right) = \frac{r_{k,k+1}}{D(s_k)} \cdot \left(e^{-D(s_k)\cdot a_k} - e^{-D(s_k)\cdot b_k}\right) \cdot \mathsf{P}_\tau\left(C\left(\ldots, s_k\right)\right)$$

Let $\pi$ be the steady state distribution of $\mathbf{O}$. The value of the probability $\mathsf{P}_\pi(C(p))$ is the portion of time spent by following the paths in the cylinder set $p$ on the long run. In the following we will use $\mathsf{P}_s$ to denote $\mathsf{P}_\tau$, with $\tau(s) = 1$.

### 3.2 Actual and Potential Checks of Quantitative Policies

Here we define our stochastic security policies through a variant of Continuous Stochastic Logic (CSL) [20,8], that extends CTL. Our logic, called $CSL_S$, comprises path formulas and state formulas. Path formulas denote measurable unions of cylinder sets, while state formulas are propositions, the atoms of which constraint the given measure $\mathsf{P}_\tau$.

**Definition 3.3** *State and path formulas are defined by:*

- *State formulas:* $\upsilon, \upsilon' ::= tt \mid \curvearrowright^a \mid \neg\upsilon \mid \upsilon \wedge \upsilon' \mid \upsilon \vee \upsilon' \mid \mathcal{C}_{\leq c}(\iota)$
- *Path formulas:* $\iota, \iota' ::= X^I\upsilon \mid \upsilon U^I \upsilon'$

The semantics of formulas is defined below over the given CTMC $\mathbf{O}$. Path formulas are evaluated over paths and state formulas are evaluated over states. Informally, $\mathcal{C}_{\leq c}(\iota)$ states that, on the long run, the portion of time spent doing any of the paths denoted by $\iota$ is bound by $p$. For simplicity we only use a $\leq$ bound, but of course we could add at no cost any other symbol of comparison, e.g. $>, \geq$. Moreover, since we focus on transitions rather than on states, in $CSL_S$ we "label" states with their outgoing transition using a class of predicates $\curvearrowright^a, a \in \mathbb{A}$.

The operator *next* $X^I\upsilon$ describes paths that start with a transition leading to a state where $\upsilon$ holds, and with duration in the interval $I$. The *until* operator $\upsilon U^I \upsilon'$ describes paths along states where $\upsilon$ does not hold until a transition leads to a state where $\upsilon'$ holds after a time in the interval $I$.

**Definition 3.4** *The semantics of state formulas is evaluated over states $s \in S$ of $\mathbf{O}$; below let $Prb(s, \iota) = \mathsf{P}_s\left(\{\sigma \mid \sigma \models \iota\}\right)$:*

| | | | |
|---|---|---|---|
| $s \models \mathrm{tt}$ | *always true* | $s \models \upsilon \wedge \upsilon'$ | *iff $s \models \upsilon$ and $s \models \upsilon'$* |
| $s \models \neg\upsilon$ | *iff $s \not\models \upsilon$* | $s \models \mathcal{C}_{\leq c}(\iota)$ | *iff $\pi_s \times Prb(s, \iota) \lhd p$* |
| $s \models \curvearrowright^a$ | *iff a transition from $s$ labelled $a$ exists* | | |

*Path formulas are evaluated over the paths of $\mathbf{O}$:*

$$\sigma \models X^I\upsilon \quad \textit{iff } \sigma[1] \textit{ is defined } \wedge\, \sigma[1] \models \upsilon \wedge \delta(\sigma, 0) \in I$$
$$\sigma \models \upsilon U^I \upsilon' \quad \textit{iff } \exists t \in I. \sigma@t \models \upsilon' \wedge (\forall t' \in [0, t). \sigma@t' \models \upsilon)$$

As proved in [20], the set $\{\sigma \mid \sigma \models \iota\}$ turns out to be measurable, hence $\iota$ *denotes* a measurable set of paths (this also implies that the definition of $Prb$ is correct).

We briefly comment on the definition for $\mathcal{C}_{\leq c}(\iota)$, recalling that $\iota$ represents a set of paths, $\pi_s$ the portion of time spent in state $s$ on the long run, while $Prb(s, \iota)$ is the probability, once in $s$, of doing a path belonging to $\iota$. Their product gives the portion of time spent doing a path denoted by $\iota$ on the long-run.

We do not present here a procedure for verifying a state formula. We refer the interested reader to [8], that gives a fixpoint characterization of $Prb$.

We define now quantitative policy $\psi$ as a $CSL_S$ formula of the form

$$\psi = \mathcal{C}_{\leq c}(\iota) \text{ with operator } \mathcal{C} \text{ no longer occurring in } \iota$$

A policy of this form endows a path formula $\iota$, denoting a measurable set of undesired paths, wrapped up by the operator $\mathcal{C}_{\leq c}$. Obviously $c$ bounds the probability of all paths denoted by $\iota$.

Summing up, with the above definitions we can explain how actual and potential checks works. The actual check requires an execution monitor that watches the computation and aborts it whenever the generated timed history is about to fall in the set described by $\iota$. This kind of monitoring causes a performance degradation because it should be always enabled.

The potential control is done by checking the semantics $P[\![h]\!]$ of a process $h$, the model, against the policies to be obeyed. We say that $h$ respects all policies occurring in it if and only if for all sub-expressions of the form $\psi_i[h_i].\psi_i[h_i] \models \psi$. Needless to say, the generation of the CTMC and of its steady state can be easily done by following the semantic definitions of Section 2.3 and by using standard packages for numerical computations. The verification is then completed by a suitable combination of the algorithms in [8] with standard model checking tools [25].

We now suggest a complementary usage of the two different ways of verifying policies put forward above. The result of a potential check can be interpreted as a bound on the probability of a monitor intervention. Indeed, suppose that $\mathcal{C}_{\leq c}(\iota)$ is verified true. The system will then execute an offending run, belonging to $\iota$, in a percentage of its time smaller than $c$. A user can consciously decide to activate the run-time monitor, based on this information, as well as on the risks that a possible violation may cause. If the risk is acceptable, the user can instead deactivate the monitor, so freeing the system from the induced overhead. Additionally, as potential analysis bounds the time spent in unsafe computations, one can evaluate the performance and reliability of his system, by bounding the time lost in computations that will be aborted. Finally, by decreasing the value of the parameter $c$, we can determine the minimum value for $\mathcal{C}_{\leq c}$ to be true — and give hints to the designer on the parts of the system need security-improving refinements.

## 4 A Working Example

A shop in Milan, called *Vestiti*, is part of dress brand chain. The shop database is mirrored in two servers: one in Milan and one in Rome. The shop is connected

to both: to the one in Milan through a private MAN and to the one in Rome through Internet. The shop can communicate with the server in Milan with low latency, but sometimes it could happen that the one in Rome performs better.

When an item is sold, the shop updates a single remote database, as they are autonomously mirrored. The manager of *Vestiti* requires the payment system to ask, at the moment of a payment, both servers and then to choose the fastest to answer. Then a cash transaction occurs: the client inserts her fidelity card; the payment system asks for offers reserved to that client; and a remainder of the offer is printed at the top of the receipt. Then the system asks both servers for a new transaction to update the database with the items sold. This race is won by the first server that answers.

Assuming as given the set of actions and rates in Table 3, we formalize the above as follows:

$$h_{Vestiti} = \Big((\mathrm{CIC},1)\cdot(\mathrm{RO},2)\cdot(\mathrm{POT},3)\cdot\psi\Big[((\mathrm{RTM},2)\cdot(\mathrm{BMT},1)+(\mathrm{RTI},1.5)\cdot(\mathrm{BIT},1))\cdot(\mathrm{DT},2)\cdot(\mathrm{ET},1)\Big]\Big)^{*}\,\delta$$

**Table 3.** Events and their description

| Event | Description | Event | Description |
|---|---|---|---|
| CIC,1 | Customer insert card | RO,2 | Ask for offers remotely |
| POT,3 | Print offer on the ticket | RTM,2 | Request MAN transaction |
| RTI,1.5 | Request Internet transaction | BIT,1 | Opening Internet transaction |
| BMT,1 | Opening MAN transaction | DT,2 | Executing transaction operations |
| ET,1 | Closing transaction | | |

However, the CEO of the dress brand is scared by using an Internet connection, that he considers much more unreliable than their own MAN. To stay on the safe side, the CEO asks the manager of *Vestiti* to enforce a security policy, so to abort internet transaction lasting more than three seconds.

The policy $\psi$ expressing the requirement of the manager is formally rendered by the $CSL_S$ expression:

$$\psi = \mathcal{C}_{\leq 0.01}\left(\neg(\curvearrowright^{BMT} \vee \curvearrowright^{CIC})\, U^{[3,\infty]}\ \curvearrowright^{CIC}\right)$$

This policy states that we require a system not to spend more than 1% portion of time doing an Internet transaction longer than 3 seconds. In other words, only 1% of computational time will be spent by a run that the security monitor will abort.

After some easy calculations, involving the computation of the steady state distribution $\pi$ of the CTMC associated with $h_{Vestiti}$, we obtain that

$$\psi[\ldots] \models \mathcal{C}_{\leq 0.01}(\neg(\curvearrowright^{BMT} \vee \curvearrowright^{CIC})\, U^{[3,\infty]}\ \curvearrowright^{CIC}) \text{ iff}$$
$$\pi_{S4} \times Prb(\psi[\ldots], \neg(\curvearrowright^{BMT} \vee \curvearrowright^{CIC})\, U^{[3,\infty]}\ \curvearrowright^{CIC}) \leq 0.01$$

Now $Prb(\psi[\ldots], \neg(\curvearrowright^{BMT} \vee \curvearrowright^{CIC})\, U^{[3,\infty]}\ \curvearrowright^{CIC}) = 0.12$ and $\pi_{S4} = 0.06$ hence, the policy is respected, because $0.06 \times 0.12 = 0.0072 \leq 0.01$.

Then, our analysis shows that the manager of *Vestiti* did a good job: the payment system of his shop always uses the fastest server available at the moment. The static check guarantees that system is quite reliable even without a security monitor switched on, because there is a low probability of violating the policy, i.e. 0.0072%. If the manager still feels unsecure and activates a security monitor, we can estimate that in a period of one hour, approximately less that 30 seconds are lost serving a payment that will result in a security exception.

## 5 Conclusions

We addressed the problem of expressing and enforcing non-functional security policies on programs. In particular we focused on quantitative security policies which constraint program behaviour over time. Our approach is based on the stochastic process algebra HE$\mu$ to abstract programs behaviour. The calculus endows the binary Kleene star iteration operator and a full-fledged sequentialization operator. The semantics of HE$\mu$ has been given in terms of CTMC using the approach of [17,16]. Security policies are expressed as formulae of $CSL_S$ predicates over CTMCs. We plan to integrate our quantitative security policies in the language-based security framework of [12,11,10,9]. In this approach programs are typed as functions with a side effect that abstractly describes the possible run-time executions of the program. Security policies are properties over effects and model-checking techniques are used to control statically whether or not the program satisfied the security policies on demands. We plan to exploit HE$\mu$ to represent quantitative effects of programs.

**Acknowledgement.** The authors would like to thank the anonymous referees for their comments that guided us to improve the quality of the paper.

## References

1. Aceto, L., Hennessy, M.: Termination, deadlock and divergence. In: Mathematical Foundations of Programming Semantics, pp. 301–318. Springer, Heidelberg (1989)
2. Ash, R., Doléans-Dade, C.: Probability and measure theory. Academic Press, London (2000)
3. Athreya, K., Lahiri, S.: Measure theory and probability theory. Springer-Verlag New York Inc. (2006)
4. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. ACM Transactions on Computational Logic (TOCL) 1(1), 170 (2000)
5. Baeten, J.: Process algebra with explicit termination. Tech. rep. (2000)
6. Baeten, J., Corradini, F.: Regular expressions in process algebra. In: Proceedings of 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005, pp. 12–19 (2005)
7. Baeten, J., Weijland, W.: Process algebra. Cambridge University Press, Cambridge (1990)
8. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.: Model-checking algorithms for continuous-time Markov chains. IEEE Transactions on Software Engineering 29(6), 524–541 (2003)

9. Bartoletti, M., Degano, P., Ferrari, G.L.: Types and effects for secure service orchestration. In: CSFW, pp. 57–69. IEEE Computer Society, Los Alamitos (2006)
10. Bartoletti, M., Degano, P., Ferrari, G.: Planning and verifying service composition. Journal of Computer Security 17(5), 799–837 (2009)
11. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Semantics-based design for secure web services. IEEE Trans. Software Eng. 34(1), 33–49 (2008)
12. Bartoletti, M., Degano, P., Ferrari, G., Zunino, R.: Local policies for resource usage analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) 31(6), 23 (2009)
13. Bergstra, J., Ponse, A., Smolka, S.: Handbook of process algebra. Elsevier Science Ltd., Amsterdam (2001)
14. Bernardo, M., Gorrieri, R.: Extended markovian process algebra. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 315–330. Springer, Heidelberg (1996)
15. Blute, R., Desharnais, J., Edalat, A., Panangaden, P.: Bisimulation for labelled Markov processes (1997)
16. Cardelli, L., Mardare, R.: The measurable space of stochastic processes. In: QEST, pp. 171–180. IEEE Computer Society, Los Alamitos (2010)
17. Cardelli, L., Mardare, R.: Stochastic pi-calculus revisited (2010) (unpublished), http://lucacardelli.name
18. Clark, G., Gilmore, S., Hillston, J.: Specifying performance measures for PEPA. In: Katoen, J.-P. (ed.) AMAST-ARTS 1999, ARTS 1999, and AMAST-WS 1999. LNCS, vol. 1601, pp. 211–227. Springer, Heidelberg (1999)
19. De Nicola, R., Latella, D., Loreti, M., Massink, M.: Rate-Based Transition Systems for Stochastic Process Calculi. Automata, Languages and Programming (2009)
20. Desharnais, J., Panangaden, P.: Continuous stochastic logic characterizes bisimulation of continuous-time Markov processes. Journal of Logic and Algebraic Programming 56(1-2) (2003)
21. Fokkink, W.: Axiomatizations for the perpetual loop in process algebra. In: Automata, Languages and Programming, pp. 571–581
22. Fokkink, W., Zantema, H.: Basic process algebra with iteration: Completeness of its equational axioms. The Computer Journal 37(4), 259 (1994)
23. Hillston, J.: A compositional approach to performance modelling. Cambridge Univ. Pr., Cambridge (1996)
24. Kemeny, J., Snell, J., Knapp, A.: Denumerable markov chains. Springer, Heidelberg (1976)
25. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Computer Performance Evaluation: Modelling Techniques and Tools (2002)
26. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
27. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
28. Norris, J.: Markov chains. Cambridge Univ. Pr., Cambridge (1998)
29. Panangaden, P.: Labelled Markov Processes. Imperial College Press, London (2009)
30. Priami, C.: Stochastic $\pi$-calculus. The Computer Journal 38(7), 578 (1995)
31. Sangiorgi, D.: On the origins of bisimulation and coinduction. ACM Transactions on Programming Languages and Systems (TOPLAS) 31(4), 1–41 (2009)

# A Formal Programming Model of Orléans Skeleton Library

Noman Javed and Frédéric Loulergue

LIFO, University of Orléans, France
Firstname.Lastname@univ-orleans.fr

**Abstract.** Orléans Skeleton Library (OSL) is a library of parallel algorithmic skeletons in C++ on top of MPI. It provides a structured approach to parallel programming. Skeletons in OSL are based on the bulk synchronous parallelism model. In this paper we present a formal semantics of OSL: its programming model formalised with the Coq proof assistant.

**Keywords:** parallel programming, algorithmic skeletons, formal programming model, the Coq proof assistant.

## 1 Introduction

If parallel architectures are now widespread, it is not yet the case for parallel programming. For distributed memory or shared memory machines, quite low level techniques such as PThreads or MPI are still widely used. To ease the programming of parallel machines, more structured approaches are needed. Algorithmic skeletons [8,9,23,24] that can be seen as higher-order functions implemented in parallel, offer a programming style in which the user combines patterns of parallel algorithms to build her application. Bulk synchronous parallelism [27,22,25,5] is another structured approach that provides a simple and portable performance model.

Our Orléans Skeleton Library [16] is a library of data parallel algorithmic skeletons that follows the BSP model. OSL is a library for the C++ language and it uses expression template techniques as an optimisation mechanism.

In order to make this kind of library more reliable, and to be able to prove the correctness of programs written in OSL, we plan to provide formal semantics of OSL. In this context we need two semantics: one for the *programming model* of OSL (i.e. the semantics that is presented to the user of the library), and one for the *execution model* of OSL (i.e. the semantics of the implementation of the library). We we will prove the equivalence of these semantics using the Coq proof assistant [26,3]: this will increase the confidence in the correctness of the implementation of OSL.

In this paper we first present informally the OSL library (section 2) before we compare this work with related papers (section 3). Then we describe the formal programming model of OSL using the Coq proof assistant (section 4) before we conclude (section 5). We also give a tiny introduction to the Coq proof assistant in appendix (section A).

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 40–52, 2011.

## 2 An Overview of Orléans Skeleton Library

Orléans Skeleton Library is a library for the C++ language that provides a set of data-parallel algorithmic skeletons. The underlying communication library is currently MPI. The implementation of OSL takes advantage of the expression templates technique [28] to be very efficient yet allowing programming in a functional style.

### 2.1 Bulk Synchronous Parallelism

From the point of view of the user, parallel applications are developed by building the appropriate combination of skeletons. All the skeletons of OSL are bulk synchronous parallel (BSP) programs.

In the BSP model, the number of memory-processor pairs is fixed during execution. These pairs are interconnected in such a way that point-to-point communications are possible. A global synchronisation unit is available in a BSP computer. The execution of a BSP program is a sequence of super-step, each one being composed of a phase where each processor computes using only the data it holds, a phase where processors exchange data and a synchronisation barrier that guarantees the completion of data exchange before the start of a new super-step.

The programmer has also access to the BSP parameters: osl::bsp_p (number of processor-memory pairs), osl::bsp_g (network bandwidth), osl::bsp_l (synchronisation time), and osl::bsp_r (processors computing power). These parameters could be obtained by benchmarking.

### 2.2 OSL Skeletons

OSL programs are similar to sequential programs but operate on a distributed data structure called distributed array. At the time of the creation of the array, data is distributed among the processors. Distributed arrays are implemented as a template class DArray<T>. A distributed array consists of bsp_p partitions, each being an array of elements of type T.

Figure 1, gives an informal semantics for the main OSL skeletons together with their signatures. In this figure, bsp_p is noted $p$. A distributed array of type DArray<T> can be seen "sequentially" as an array $[t_0, \ldots, t_{t.size-1}]$ where $t.size$ is the global size of the distributed array $t$.

The two first skeletons are the usual combinators to apply a function to each element of a distributed array (resp. of two distributed arrays). The first argument of both map and zip could be either a pointer function, or a C++ functor either extending std::unary_function or std::binary_function. There are two variants: mapIndex and zipIndex. Each of these variants takes a function with an additional argument: an integer representing the global index of each element in the distributed array.

It is possible to expose how a distributed array is distributed among the processors using the getPartition skeleton. It transforms a distributed array of

| Skeleton | Signature<br>Informal semantics |
|---|---|
| map | DArray<W> map(W f(T), DArray<T> t)<br>map($f$, $[t_0, \ldots, t_{t.size-1}]$) = $[f(t_0), \ldots, f(t_{t.size-1})]$ |
| zip | DArray<W> zip(W f(T,U), DArray<T> t, DArray<U> u)<br>zip($f$, $[t_0, \ldots, t_{t.size-1}]$, $[u_0, \ldots, u_{t.size-1}]$) = $[f(t_0, u_0), \ldots, f(t_{t.size-1}, u_{t.size-1})]$ |
| reduce | <T> reduce(T⊕(T,T), DArray<T> t)<br>reduce($\oplus$, $[t_0, \ldots, t_{t.size-1}]$) = $t_0 \oplus t_1 \oplus \ldots \oplus t_{t.size-1}$ |
| getPartition | DArray<Vector<T> > getPartition(DArray<T> t)<br>getPartition($[t_0, \ldots, t_{t.size-1}]$) = $[\ [t_0^0, \ldots, t_{l_0-1}^0]\ , \ldots,\ [t_0^{p-1}, \ldots, t_{l_{p-1}-1}^{p-1}]\ ]$ |
| flatten | DArray<T> flatten(DArray<Vector<T> > t)<br>flatten($[\ [t_0^0, \ldots, t_{l_0-1}^0]\ , \ldots,\ [t_0^{p-1}, \ldots, t_{l_{p-1}-1}^{p-1}]\ ]$) = $[t_0, \ldots, t_{t.size-1}]$ |
| permute | DArray<T> permute(**int** f(**int**), DArray<T> t)<br>permute($f$, $[t_0^0, \ldots, t_{l_0}^0]$) = $[t_{f^{-1}(0)}^0, \ldots, t_{f^{-1}(l_0-1)}^0]$ |
| shift | DArray<T> shift(**int** dec, T f(T), DArray<T> t)<br>shift($d$, $f$, $[t_0^0, \ldots, t_{l_0}^0]$) = $[f(0), \ldots, f(d-1), t_0, \ldots, t_{t.size-1-d}]$ |
| bcast | DArray<T> bcast(DArray<T> t)<br>bcast $[\ t_0^0, \ldots, t_{l_0-1}^0,\ \ldots\ , t_0^{p-1}, \ldots, t_{l_{p-1}-1}^{p-1}\ ]$ = $[\ t_0^0, \ldots, t_{l_0-1}^0,\ \ldots\ , t_0^0, \ldots, t_{l_0-1}^0\ ]$ |
| balance | DArray<T> balance(DArray<T> t)<br>balance $[t_0, \ldots, t_{t.size-1}] = [t_0, \ldots, t_{t.size-1}]$ |
| gather | DArray<T> gather(DArray<T> t)<br>gather $[t_0, \ldots, t_{t.size-1}] = [t_0, \ldots, t_{t.size-1}]$ |

**Fig. 1.** OSL Skeletons

type DArray<T> into a distributed array of type DArray<Vector<T> > and of size bsp_p. In the resulting distributed array, each processor contains only one element: a C++ vector, the former partition on this processor. The flatten skeleton is the inverse operation of getPartition.

Parallel reduction with a binary *associative* operator $\oplus$ is performed using the reduce skeleton. Communications are needed to execute a reduce. permute and shift are also communication functions. permute redistributes the array, according to a function f, bijective on the interval $[0, t.size-1]$. The shift skeleton allows to shift elements on the right (the case shown in the figure) or the left depending on the sign of its first argument. The missing values, at the beginning or the end of the array, are given by function f. bcast copies the content of the root processor to all other processors.

The two next functions only modify the distribution of the distributed array, not its content. balance distributes evenly the content of the distributed array if it is not evenly distributed. gather gathers the content of the distributed array at root processor, the partitions on the remaining processor become empty.

## 3 Related Work

There exist many algorithmic skeleton libraries and languages: [14] is a recent survey. Here we focus on work on formal semantics. Eden [20] and BSML [12,21] are two parallel functional languages that are often used for implementing algorithmic skeletons. The former has a formal semantics on paper [15] whereas the

latter has a *mechanised* formal semantics [6]. Moreover a new algorithmic skeleton called BH as been implemented and its implementation proved correct using the Coq proof assistant. This BH skeleton is used in a framework for deriving programs written in BH from specification [13].

Some other work focus on algorithmic skeleton libraries, to our knowledge none is formalised and the properties of the semantics verified using a proof assistant. [10] presents a data-parallel calculus of multi-dimensional arrays, but it is a formal semantics without any related implementation. The Lithium [2] algorithmic skeleton library for Java differs from OSL as it is stream-based. [1] proposes in a single formalism a programming model and a (high-level) execution model for Lithium. The skeletons of [11] are also stream-based but the semantics is used rather as a guideline for the design of the meta-programmed optimisation of the skeletons in C++.

The semantics of the Calcium library is described in [7] and further extended in a shared memory context to handle exceptions [19]. In [7], the focus in on a programming model semantics (operational semantics) as well as a static semantics (typing) and the proof of the subject reduction property (the typing is preserved during evaluation). In this work the semantics of the skeletons are detailed, but not the semantics of what the authors call the "muscles" i.e. the sequential arguments of the skeletons (the semantics of the host language of the library, in the particular case Java). The set of skeletons of Calcium includes a set of task parallel skeletons, which contains, among others, skeletons that give a sequential control but at the global level of all the parallel program. These skeletons are parallel because their branches or bodies are parallel (conditionals and while/for loops). In OSL we mix the skeletons with the usual constructs of the host C++ language to write the sequential control flow at the global level of the parallel program. The remaining skeletons in Calcium are data-parallel skeletons including map, and divide-and-conquer skeletons. The map skeleton, for example, is however different from our map. The OSL map is more similar to map functions in functional programming as it takes two arguments: a function $f$ to be applied to each element of the collection $l$ which is the second argument. In functional programming this collection is a list, in OSL it is a distributed array. In Calcium the map skeleton takes two additional functions: one that describes how the input collection is cut into pieces and another function that describes how the pieces (obtained by applying $f$ to the previous pieces) are combined together to form the output collection.

## 4 OSL Mechanised Semantics: Programming Model

We now present how we modelled the programming model of OSL using the Coq proof assistant. We first explain how the modelling of the data structure of distributed arrays and of the syntax is done. We then present a big-step semantics and its properties.

### 4.1 Distributed Arrays

First of all we need to model the parallel data structure of our OSL library: the distributed arrays. The content of a distributed array can be seen as a usual sequential array plus information about its distribution. In Coq we model the content of the arrays by lists. The distribution is modelled by a data structure similar to lists but with the size of the collection inside the type: vectors. A vector of type vector A n has size n and contains values of type A. A distribution is a vector of *natural numbers*: each natural number is the number of elements per processor. The size of a vector of distribution is bsp_p, the number of processor of the BSP machine. bsp_p is strictly positive. To cleanly formalise the fact that the syntax and semantics are parametrised by the number of processors of the parallel machine, the semantics is a functor, i.e. a module that takes as argument another module. This argument module has the following type:

```
Module Type BSP_PARAMETERS.
  Parameter lastProcessor : nat.
End BSP_PARAMETERS.
```

lastProcessor is supposed to be the processor identifier of the last processor. We then define:

```
Definition bsp_p := S Bsp.lastProcessor.
```

This allows to instantiate the functor with a module containing a specific value for lastProcessor in order to write examples and execute our semantics within Coq.

The type of distributed array is a record type:

```
Record distributedArray (A:Type) := mkDistributedArray {
  distributedArray_data : list A;
  distributedArray_distribution: vector nat bsp_p;
  distributedArray_invariant:
    List.length distributedArray_data = sum distributedArray_distribution
}.
```

This type contains the two fields already described: the content of the parallel vector (distributedArray_data), and the distribution of this content on the processors (distributedArray_distribution).

However there is a third field: a proof that the two fields form indeed a coherent representation of a distributed array. The sum of the elements of the distribution (computed using the function sum, omitted here) should be the length of the content list.

Values of this type are a kind of inner representation of distributed arrays that the user of the Orleans Skeleton Library could not used directly. She will be given a syntax for writing OSL programs.

### 4.2 Syntax and Typing

As in [7], we would like to model the semantics of our library without being obliged to model the whole syntax of the host language. The language of the

Coq proof assistant can be seen as a pure functional programming language plus ways to express logical properties. Therefore the sequential values and functions of the host language (here C++) can be written as Coq values and functions. In the case of functions we thus model only their input/output behaviour.

The result of a computation, a value, could be either a usual sequential value, for example the result of the application of the reduce skeleton, or a distributed array, for example the result of the application of the map skeleton.

There are several ways of formalising the syntax of OSL programs. We shall illustrate this by two short examples dealing only with the construct for distributed arrays and the map skeleton. The first solution follows:

```
Inductive expression :=
| DistributedArray: ∀A:Type, list A →expression
| Map : ∀A B, (A→B) →expression →expression
```

To simplify the example, a distributed array is just modelled as a list of values. All values being typed in Coq, the constructor for this case of the inductive type expression should also take as argument the type of the elements of the list. For the Map constructor, the first argument is the "muscle" argument, the function $f$ to be applied to each element of the distributed array, the second expression. Here again the input and output types of the function should be given.

This grammar however models possibly ill-typed expressions of our language of skeletons. It is possible to define the following Coq term:

```
Definition e : expression :=
  Map string string (append "!") (DistributedArray nat [1;2;3]).
```

In Coq it is possible to indicate than some arguments may be implicit: it is the case here for the types arguments of the two constructors Map and DistributedArray and we could write:

```
Definition e : expression := Map (append "!") (DistributedArray [1;2;3]).
```

The expression e is well typed for Coq but it *represents* an ill-typed expression of our skeleton language as the muscle function append operates on strings instead of natural numbers. We could as in [7] define a type system and prove that the operational semantics we will define follows the subject reduction property (i.e. it preserves the typing).

However there is another solution: we could model the grammar in such a way that only well-typed (in the skeleton language point of view) expressions could be modelled in Coq:

```
Inductive typedExpression (A:Type) :=
| TDistributedArray : list A →typedExpression A
| TMap: ∀B, (B→A) →typedExpression B →typedExpression A.
```

Here the grammar is typed. An expression of type typedExpression A represents an expression whose value is a distributed array whose elements have type A. The expression e could not be defined in Coq as a typedExpression: the input

```
Inductive seqExpr : Type →Type :=
| SeqValue: ∀A, A →seqExpr A
| Reduce: ∀A, seqExpr (A→A→A) →seqExpr A →parExpr A →seqExpr A
| SeqApply: ∀A B, seqExpr (A→B) →seqExpr A →seqExpr B
with parExpr : Type →Type :=
| ParValue: ∀A, distributedArray A →parExpr A
| Replicate: ∀A, seqExpr A →seqExpr nat →parExpr A
| Init: ∀A, seqExpr (nat→A) →seqExpr nat →parExpr A
| CreateAtRoot: ∀A, seqExpr (list A) →parExpr A
| Map: ∀A B, seqExpr (A→B) →parExpr A →parExpr B
| Zip: ∀A B C, seqExpr (A→B→C) →parExpr A →parExpr B →parExpr C
| MapIndex: ∀A B, seqExpr (nat→A→B) →parExpr A →parExpr B
| ZipIndex: ∀A B C, seqExpr (nat→A→B→C) →parExpr A →parExpr B →parExpr C
| Shift: ∀A, seqExpr Z →seqExpr(nat→A) →parExpr A →parExpr A
| GetPartition: ∀A, parExpr A →parExpr(list A)
| Flatten: ∀A, parExpr(list A) →parExpr A
| Permute: ∀A, seqExpr (nat→nat) →parExpr A →parExpr A
| Balance: ∀A, parExpr A →parExpr A
| Gather: ∀A, parExpr A →parExpr A
| Bcast: ∀A, parExpr A →parExpr A.

Inductive expr : Type →Type :=
| Seq: ∀A, seqExpr A →expr A
| Par: ∀A, parExpr A →expr (distributedArray A).
```

**Fig. 2.** OSL Syntax in Coq

type of the muscle function in the Map constructor should be the type of the elements of the second argument of Map.

Therefore by defining the operational semantics by a function or a relation that relates only expressions that represent skeleton expressions of the same type, then we have the subject reduction for free.

The syntax of OSL is actually a bit more complicated as we distinguish between expressions whose values have a sequential type and expressions whose values have parallel types, these two kinds of expressions being mutually recursive. The whole syntax is in figure 2. In order to be able to apply a "sequential" program to the result of the evaluation of a skeleton expression, we provide a SeqApply constructs. The SeqValue constructors is simply used to provide "muscles" to the skeletons.

The three first Coq constructors of the parExpr type are the usual OSL C++ class constructors: we can build a distributed array by specifying its size and a value that will be replicated everywhere (Replicate), or the content of the distributed array could be specified by a function from array indices to values (Init). In these two cases, the data is distributed evenly on the processors. The third constructor is used to build a distributed array containing values only at the root processor (CreateAtRoot). The other Coq constructors model the skeleton informally presented in section 2.

### 4.3 Big-Step Semantics

For the formalisation of the big-step semantics of OSL, we define three functions, the two first begin mutually recursive:

- seqEvaluation: forall A : Type, seqExpr A → result A
- parEvaluation: forall A : Type, parExpr A → result (distributedArray A)
- evaluation: forall A : Type, expr A → result A

The result type is used in a monadic style [29] in order to model possible errors during evaluation, without being too cumbersome to use compared to a solution with optional values and pattern-matching. As in [18] for example, we use a convenient Coq feature that allows to define notations:

```
Inductive result (A: Type) : Type :=
| Ok: A →result A
| Error: string →result A.

Definition bind (A B: Type) (f: result A) (g: A →result B) : result B :=
  match f with
  | Ok x ⇒g x
  | Error msg ⇒Error msg
  end.

Notation "'do' X <− A ; B" := (bind A (fun X ⇒B)).
```

With this notation, the big-step semantics functions are quite readable. For example the case for the evaluation of the reduce skeleton in the seqEvaluation function is written as follows:

```
| Reduce A op neutral pe ⇒
    do op <− seqEvaluation op;
      do neutral <− seqEvaluation neutral;
        do da <− parEvaluation pe ;
          Ok(List.fold_right op neutral (distributedArray_data da))
```

We first evaluate the "muscles" of the skeletons. If one of these calls raises an error, then the function immediately returns this error, otherwise it binds the obtained value with the variable before the <− arrow and continues to evaluate the expression after the ;.

The parEvaluation function produces values of type distributedArray. In order to keep this function short, we defined auxiliary functions that transforms distributed arrays. The parEvaluation function thus first recursively calls itself and seqEvaluation on the arguments of the expression it evaluates, and obtains *values*, in particular in the parallel case, values of type distributedArray. Then it calls the appropriate auxiliary function. For example:

```
| Replicate _ se se' ⇒
    do v <− seqEvaluation se;
      do size <− seqEvaluation se';
        Ok (replicate v size)
```

The replicate function, and all the auxiliary functions, are defined using the Program feature of Coq:

```
Program Definition replicate(A:Type)(value:A)(size:nat) : distributedArray A :=
  mkDistributedArray
  (List.map (fun index⇒value) (List.seq 0 size))
  (evenDistribution size)
  _.
Next Obligation.
  autorewrite with length; rewrite sumEvenDistribution; trivial.
Defined.
```

For building a value of type distributedArray, we need three components:

- the content of the distributed array, in this case it is defined on the third line (we apply a constant function to all the elements of a list of natural numbers, of the specified size),
- the distribution, in this case it is defined on the fourth line, by a call to the function evenDistribution,
- a *proof* that the content and the distribution are coherent.

The two first components are written very similarly to functional programs. For the proof however, it is easier to use the interactive proof mode. Thus we do not give this third component: we use the wildcard _ instead. Coq then generates proof obligations that should be proved in order for the value replicate to be defined. The proof is here quite simple because most of the work is done in the lemma sumEvenDistribution that it itself proved using several other lemmas.

This replicate function could not directly raise an error. Few skeletons can: the zip skeleton if the two parallel arguments do not have the same distribution, the permute skeleton if the function in argument is not bijective, and the flatten skeleton if the distribution of its argument is not one element (of type list) per processor.

By construction the type of the expressions are preserved during evaluation: we have subject reduction for free.

evaluation, seqEvaluation and parEvaluation are functions. They can be applied to OSL program examples in Coq. The results of such evaluations can be output. This allows to design and implement automatic tests to check if the formal semantics and the implementation are coherent. This can serve to debug both: the formal semantics may be erroneous because we were wrong in the modelling, or the implementation may contain bugs.

All the Coq source code of this formalisation is available at:

`http://traclifo.univ-orleans.fr/OSL`.

## 5 Conclusion and Future Work

In this paper we have presented a formal semantics of the programming model of the Orléans Skeleton Library, modelled using the Coq proof assistant. A formal

programming model is necessary to reason about OSL programs in Coq. Writing such a formal semantics and checking its properties using a proof assistant make necessary to look into all the details of the semantics. Based on this work we improved the reliability of the current implementation of the OSL library in C++. It is a first step: we plan to design and implement a formal semantics of the execution model and prove its equivalence with the programming model.

One limitation of this approach is that we are modelling the programs rather than trying to prove directly the code. This this mainly due to the fact that C++ is a complex programming language and, to our knowledge, there is no support for the proof of correctness of C++ programs with theorem provers or other tools. However to fill the gap between what is modelled and what is proved correct, we plan in the PaPDAS project (`http://traclifo.univ-orleans.fr/PaPDAS`) to design a skeletal parallel programming language, extension of C (not C++), and to implement and prove correct a compiler for this language, building on the CompCert compiler [17,18].

**Acknowledgements.** This work is supported by the *Agence National de la Recherche* through the project "Parallel Programming Development with Algorithmic Skeletons" (PaPDAS). Noman Javed is supported by a grant from the Higher Education Commission of Pakistan.

## References

1. Aldinucci, M., Danelutto, M.: Skeleton-based parallel programming: Functional and parallel semantics in a single shot. Computer Languages, Systems and Structures 33(3-4), 179–192 (2007)
2. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Generation Computer Systems 19, 611–626 (2003)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
4. Bertot, Y.: Coq in a hurry (2006), `http://hal.inria.fr/inria-00001173`
5. Bisseling, R.: Parallel Scientific Computation. A structured approach using BSP and MPI. Oxford University Press, Oxford (2004)
6. Bousdira, W., Gesbert, L., Loulergue, F.: Syntaxe et sémantique de Revised Bulk Synchronous Parallel ML. In: Conchon, S., Mahboubi, A. (eds.) Journées Francophones des Langages Applicatifs (JFLA), pp. 117–146. Studia Informatica Universalis, Hermann (2011)
7. Caromel, D., Henrio, L., Leyton, M.: Type safe algorithmic skeletons. In: 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), pp. 45–53. IEEE Computer Society, Los Alamitos (2008)
8. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1989), `http://homepages.inf.ed.ac.uk/mic/Pubs`
9. Darlington, J., Field, A.J., Harrison, P.G., Kelly, P., Sharp, D., Wu, Q., While, R.: Parallel Programming Using Skeleton Functions. In: Reeve, M., Bode, A., Wolf, G. (eds.) PARLE 1993. LNCS, vol. 694. Springer, Heidelberg (1993)
10. Di Cosmo, R., Pelagatti, S., Li, Z.: A calculus for parallel computations over multidimensional dense arrays. Computer Language Structures and Systems 33(3-4), 82–110 (2007)

11. Falcou, J., Sérot, J.: Formal Semantics Applied to the Implementation of a Skeleton-Based Parallel Programming Library. In: Bischof, C.H., Bücker, H.M., Gibbon, P., Joubert, G.R., Lippert, T., Mohr, B., Peters, F.J. (eds.) Advances in Parallel Computing Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, vol. 15, pp. 243–252. IOS Press, Amsterdam (2007)
12. Gava, F., Loulergue, F.: A Polymorphic Type System for Bulk Synchronous Parallel ML. In: Malyshkin, V. (ed.) PaCT 2003. LNCS, vol. 2763, pp. 215–229. Springer, Heidelberg (2003)
13. Gesbert, L., Hu, Z., Loulergue, F., Matsuzaki, K., Tesson, J.: Systematic Development of Correct Bulk Synchronous Parallel Programs. In: The 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 334–340. IEEE Computer Society, Los Alamitos (2010)
14. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software, Practrice & Experience 40(12), 1135–1160 (2010)
15. Hidalgo-Herrero, M., Ortega-Mallén, Y.: An Operational Semantics for the Parallel Language Eden. Parallel Processing Letters 12(2), 211–228 (2002)
16. Javed, N., Loulergue, F.: OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays. In: Dou, Y., Gruber, R., Joller, J.M. (eds.) APPT 2009. LNCS, vol. 5737, pp. 436–451. Springer, Heidelberg (2009)
17. Leroy, X.: Formal verification of a realistic compiler. CACM 52(7), 107–115 (2009)
18. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
19. Leyton, M., Henrio, L., Piquer, J.M.: Exceptions for algorithmic skeletons. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 14–25. Springer, Heidelberg (2010)
20. Loogen, R., Ortega-Mallen, Y., Pena-Mari, R.: Parallel functional programming in eden. Journal of Functional Programming 3(15), 431–475 (2005)
21. Loulergue, F., Gava, F., Billiet, D.: Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3515, pp. 1046–1054. Springer, Heidelberg (2005)
22. McColl, W.F.: Scalability, portability and predictability: The BSP approach to parallel programming. Future Generation Computer Systems 12, 265–272 (1996)
23. Pelagatti, S.: Structured Development of Parallel Programs. Taylor & Francis, Abington (1998)
24. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, Heidelberg (2003)
25. Skillicorn, D.B., Hill, J.M.D., McColl, W.F.: Questions and Answers about BSP. Scientific Programming 6(3), 249–274 (1997)
26. The Coq Development Team: The Coq Proof Assistant, http://coq.inria.fr
27. Valiant, L.G.: A bridging model for parallel computation. Comm. of the ACM 33(8), 103 (1990)
28. Veldhuizen, T.: Techniques for Scientific C++. Computer science technical report 542, Indiana University (2000)
29. Wadler, P.: Monads for Functional Programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995)

# A A Short Introduction to Coq

The Coq proof assistant [26,3,4] is based on the calculus of inductive constructions. This calculus is a higher-order typed $\lambda$-calculus. The Curry-Howard correspondence is at the core of Coq: Theorems are types and their proofs are terms of the calculus. The Coq system provides a language of tactics to help the user to build proof terms.

We illustrate quickly all these notions on a short example. We first define a new inductive type, the type of natural numbers in the Peano style:

```
Inductive nat : Set :=
  | O : nat
  | S : nat →nat.
```

nat has type **Set**: it is similar to a usual data-type in a functional language.

We also define the plus recursive function on natural numbers:

```
Fixpoint plus (n1 n2:nat) {struct n1} : nat :=
match n1 with
  | O ⇒n2
  | S n ⇒S(plus n n2)
end.
```

In this recursive definition we should specify which argument is structurally decreasing (n1 in the example). This is because all functions must be terminating in Coq. In both definitions, we gave the type of the new name we wanted to define as well as a term of this type. We then define a lemma:

```
Lemma plus_n_O : ∀n, plus n O = n.
Proof.
  induction n.
  (* case n=0 *) simpl. reflexivity.
  (* case n>0 *) simpl. rewrite IHn. reflexivity.
Qed.
```

If we check (using the Check command of Coq) the type of expression, we would obtain Prop. This definition is a proposition. It belongs to the logical realm. To define plus_n_O we should not only provide a type, but also a term of this type: a proof of the lemma. We could write directly such a term, but it is usually complicated. Coq provides a language of tactics to help the user to build proof terms. In the top-level of Coq, entering line beginning with **Lemma** activates the interactive proof mode. The Coq proof assistant indicates that we should prove the following goal:

```
============================
 forall n : nat, plus n O = n
```

We prove this goal by induction on n using the tactic induction n. The system indicates now two goals to prove:

```
  ============================
   plus O O = O

subgoal 2 is:
 plus (S n) O = S n
```

The first one is proved using the definition of plus using the tactic simpl which yields the goal $0 = 0$ and this case is ended by the application of the tactic reflexivity. The second one is the inductive case:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

After simplification, we obtain the goal S(plus n O) = S n. We solve it first by rewriting plus n O in n using the IHn hypothesis and then we conclude by reflexivity. Actually, Coq has some automation. The plus_n_O lemma could be proved using one tactic: auto.

Mixing logical and computational parts is possible in Coq. For example a function of type A→B with a precondition P and a post-condition Q corresponds to a constructive proof of type: ∀x:A, (P x) →exists y:B →(Q x y). This could be expressed in Coq using the inductive type sig:

```
Inductive sig (A:Set) (P:A→Prop) : Set := | exist: ∀(x:A), (P x) →(sig A P).
```

It could also be written, using syntactic sugar, as {x:A|(P x)}.

This feature is used in definition of the function pred:

```
Require Import Program.

Program Definition pred (n:nat | n<>O) : {q:nat|(S q)=n} :=
  match n with
    | O ⇒_
    | S n ⇒n
  end.
```

The specification of this function is: ∀n : {m : nat | m<>O}, {q : nat | S q = ‘n} where ‘n represents the natural number part of n (the other part being a proof that this natural number is not zero). We define pred using the **Program** feature of Coq. This feature allows the user to write a function with post-conditions as if there were no post-condition. **Program** generates proof obligations to be proved to ensure that the function result indeed meets the post-condition. Moreover in this example the proof obligations are proved automatically by the system.

# LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem

Victor E. Malyshkin and Vladislav A. Perepelkin

Institute of Computational Mathematics and Mathematical Geophysics
Russian Academy of Sciences
Novosibirsk
{malysh,perepelkin}@ssd.sscc.ru
http://ssd.sscc.ru

**Abstract.** The peculiarities of the LuNA run-time subsystem implementation are considered. LuNA is the language and system of fragmented programming. The peculiarities are conditioned by the properties of numerical algorithms, to implementation and execution of which the LuNA is mainly oriented.

**Keywords:** run-time, parallel and distributed computing, parallel programming language and system, model of parallel program.

## 1 Introduction

The idea of data and algorithms fragmentation has been exploited in programming at least since the early 1970-s [1–8]. This approach to computation organization with the use of a run-time subsystem is used if the solutions on how to execute a program or its parts, how to distribute the resources should be done dynamically. Different modifications of this approach were embodied in programming systems [2–5]. Many programming systems use the run-time subsystems for the organization of computation [5–13]. In [2] instead of commonly used run-time system for program execution, a special hardware and operating system were developed. Our project of the LuNA fragmented programming system is oriented to the creation of a parallel numerical subroutine library.

## 2 Introductory Definitions

A general model of a program in the above mentioned systems can be described as computational model [3].

### 2.1 General Model Definition

Given:

- The finite set $\mathbf{X}=\{x, y, ..., z\}$ of variables for representation of different computed values;

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 53–61, 2011.

- The finite set **F**={*a, b, ..., c*} of functional symbols (operations, Fig. 1.a), $m \geq 0$ is the number of input variables, $n \geq 0$ is the number of output variables;
- in($a$)=($x_1, ..., x_m$) is a set of input variables, out($a$)=($y_1, ..., y_n$) is a set of output variables (Fig. 1), if $i \neq j \rightarrow y_i \neq y_j \,\&\, x_i \neq x_j$

Model C=(**X**,**F**) is called simple computational model (SCM). Operation $a \in \mathbf{F}$ describes the possibility to compute the variables out($a$) from the variables in($a$), for example, with the use of a procedure. The model can be graphically depicted (fig. 1.b, 1.c)

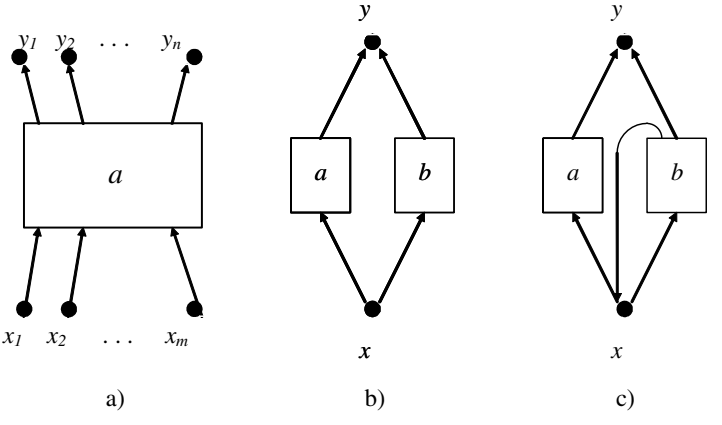



**Fig. 1.** Examples of operations, variables and models

Let $V \subseteq \mathbf{X}$, $F \subseteq \mathbf{F}$ be given. A set of functional terms $T(V,F)$ is defined as follows:

1. If $x \in V$, then $x$ is a term $t$, $t \in T(V,F)$; in($t$)={$x$}; out($t$)={$x$}.
2. Let $\{t^1, ..., t^s\} \subseteq T(V,F)$ and $a \in F$, in($a$)=($x_1,...,x_s$) be given. The term $t=a(t^1,...,t^s)$ is included into $T(V,F)$ if $\forall i(x_i \in \text{out}(t^i))$, $\text{in}(t)=\bigcup_{i=1}^{s} \text{in}(t^i)$, out($t$)=out($a$). Here $t=a(t^1,...,t^s)$ denotes that $t$ is the term $a(t^1,...,t^s)$.

A term is depicted as a tree that contains both operations and variables of the term, see Fig. 2.a.

We say that a term $t$ computes a variable y, if $y \in \text{out}(t)$. The set of terms $T(V,F)$ defines all the variables of the SCM, that can be computed from $V$ variables. A set of terms $T_V^W = \{t \in T(V,F) \mid \text{out}(t) \cap W \neq \varnothing\}$ computes all the variables from $W$ that can be computed from $V$ variables.

Any such subset $R \subseteq T_V^W$ that $\forall x \in W \exists t \in R(x \in \text{out}(t))$ is called $(V,W)$-plan and defines an algorithm computing the variables $W$ from the variables $V$. Here $V$ and $W$ denote the sets of input and output variables of the algorithm respectively. Everywhere further a set of recursively countable functional terms is considered as a representation of an algorithm.

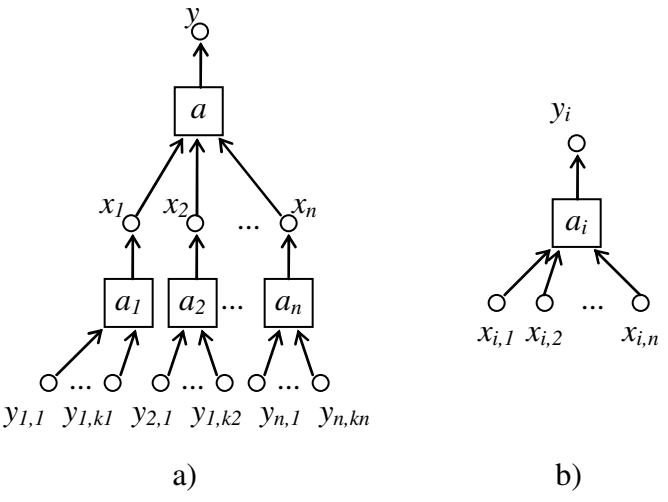



**Fig. 2.** Depicted terms

### 2.2 Interpretation

Let $V \subseteq \mathbf{X}$ be given. Interpretation $\boldsymbol{I}$ in the domain $D$ is the function that assigns to:

- every variable $x \in V$ an entry $d_x = \boldsymbol{I}(x) \in D$, $d_x$ is a value of the variable $x$ in the interpretation $\boldsymbol{I}$,
- to every operation $a \in \mathbf{F}$, $\text{in}(a)=\{x_1, x_2, ..., x_m\}$, $\text{out}(a)=\{y_1, y_2, ..., y_n\}$, a computable function $f_a$: $D^m \to D^n$,
- to every term $t=a(t_1,t_2,...,t_m)$ a superposition of the functions accordingly to the rule $\boldsymbol{I}(a(t_1,t_2,...,t_m))=f_a(\boldsymbol{I}(t_1),\boldsymbol{I}(t_2),...,\boldsymbol{I}(t_m))$.

If $t=a(t_1,t_2,...,t_m)$ is an arbitrary term, $\text{in}(a)=\{x_1, x_2, ..., x_m\}$, $\text{out}(a)=\{y_1, y_2, ..., y_n\}$, then $\boldsymbol{I}(\text{out}(a))=val(t)=(d_1,d_2,...,d_n)=f_a(valx_1(t_1),valx_2(t_2),...,valx_n(t_n))$.

Further it is assumed that for every function $f_a=\boldsymbol{I}(a)$ there exists a module (procedure) $mod_a$ that can be used in a program to compute the function $f_a$.

### 2.3 Correct Interpretation

If there exist two different terms $t_1$ and $t_2$, $y \in \text{out}(t1) \cap \text{out}(t2)$, $\text{in}(t1) \cup \text{in}(t2) \subseteq V$, then $valy(t1)=valy(t2)$ in the interpretation $\boldsymbol{I}$, the interpretation $\boldsymbol{I}$ is called correct interpretation. In the correct interpretation for any variable $y$, any pair of terms $t_1$ and $t_2$, $y \in \text{out}(t_1) \cap \text{out}(t_2)$ yields the same value, $valy(t_1)=valy(t_2)$.

For definition of mass computations this model is extended by the inclusion indexed operations and indexed variables (arrays), fig. 2.b. This technical work can be easily done. Obviously, in this extended model, a mass algorithm is represented by a potentially infinite recursively countable set of functional terms [3].

A program that implements an algorithm, represented by a set of functional terms, can be constructed with the procedure calls to $mod_a$ done in the order that does not

contradict to the information dependences between the operations imposed by the terms structures. Usually, a run-time subsystem is used to implement all the calls in the proper order.

# 3 The LuNA Fragmented Programming System

## 3.1 Model Modifications

The LuNA fragmented programming system implements the above model. In order to provide the reachability of high performance of a fragmented program execution the above model was essentially transformed [14]:

- data and computation fragmentations were included into the model [15–17]; the subsets of functional terms are aggregated into bundles, forming aggregated variables (data fragments – DF) and aggregated operations (fragments of computations – FC),
- multiple assignment variables were included, whereas every FC is permitted to be executed once only,
- the users's recommendation are introduced into an algorithm description, that are used by the LuNA run-time subsystem for improvement of the fragmented algorithm execution.

Additionally, LuNA applications are restricted by the numerical algorithms. Regular structure of mesh numerical algorithms essentially simplifies the algorithms of the LuNA run-time subsystem implementation. As result, the algorithms of LuNA run-time subsystem provide high performance execution of numerical algorithms and don't guarantee good execution of the algorithms from the another application area. The above modifications allowed essential improvement of the algorithms of the LuNA run-time subsystem implementation providing high performance of the LuNA programs. All the above modifications preconditioned the main features and peculiarities of the LuNA run-time subsystem implementation.

## 3.2 Scheme of the LuNA Implementation

LuNA program (FP – fragmented program) is constructed in two main stages:

a). an application algorithm fragmentation [15–17] and its representation by the set of fragments of computations (operations in the above model), the set of data fragments and partial order relation ρ in order to define the information dependences between fragments of computations (FC). Representation of the algorithms with the LuNA input language, including user's recommendations.

b). such a representation is already considered as an executable FP. The execution is organized in next 3 steps:

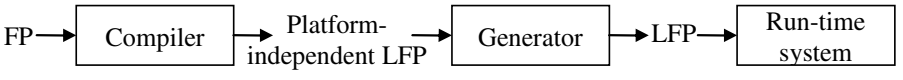



**Fig. 3.** The 3 steps of FP execution

- **Compilation.** On this step all the decisions (construction of the initial resources allocation and control), that can be done statically, are made. Also the compiler analyses the FP and proposes one or more preliminary schemes of the FP execution and resources assignment. For example, several DFs can be assigned to a one memory slot in order to save the memory; or a number of CFs can be folded into a loop by defining additional direct control. The initial resources allocation is not fully constructed on this stage in order to provide the desirable level of asynchronism in the course of the FP execution. The compiler also checks the FP in order to recognize the syntax errors and to accomplish some other traditional compilation tasks too. The compiler has no information about hardware configuration or input data. The result of the compilation is the platform-independent FP.
- **Generation.** On this step all the decisions whose making depends on the computer system architecture and its configuration are generated. The generator takes the platform-independent FP as input and a description of a certain computer system configuration. The generator defines the parameters of the FP, such as DF size, in order to fit better the hardware. It also selects one of schemes of FP execution (if the compiler has provided more than one), which better fits the given hardware configuration. The result of generation is FP, executable by the run-time subsystem.
- **Execution** is provided by the LuNA run-time subsystem. The set $T_V^W$ of the functional terms is not really constructed, the necessary term is constructed if necessary only. The run-time subsystem provides dynamic properties of the FP execution, such as dynamic workload balancing. In the course of the FP execution the run-time subsystem is capable to change the order of the FC execution and resources allocation schemes in order to optimize the efficiency of FP execution in run-time.

In order to provide the high performance of an FP execution two main problems should be solved by the LuNA run-time subsystem: to choose and to assign for execution a certain FC and to construct the rest of resources allocation.

### 3.3 LuNA Input Language

As usual [3], the LuNA input language contains the facilities for FC, DF and control $\rho$ description. Also it contains the user's recommendations, which provide the compiler and the run-time subsystem with the additional information on how to improve the execution of the FP. More detailed description of the FP representation and its peculiarities are considered in [14].

Consider an example of DF, FC and ρ definitions, written with the input LuNA language:

LuNA run-time subsystem utilizes the following types of recommendations.

**Priority.** Priority is a real-valued function, defined on the set of FCs. If there is an ability for run-time subsystem to fetch for execution a number different FCs, then the ones with the highest priority are fetched. Definition of the priorities allows controlling the flow of FCs execution in order to optimize performance (see profiling subsection below). Like the rest recommendations, execution of the FP according to the priorities is not mandatory. Depending on different factors of a certain situation, run-time subsystem can accept or not the defined recommendations.

**Table 1.** Example of DF, FC and ρ definitions in LuNA language

| **Definitions of a FP** | **Description** |
|---|---|
| df x[i] : block(real, M) \| i=1..N; | Definition of N DFs x[i], each containing M entities of real type. |
| cf a[i] : func_a(in: x[i]; out: y[i], z[i]) \| i=1..N; | Definition of N FCs a[i] with specified input and output DFs. For implementation of FCs a[i] the func_a procedure is assigned. |
| a[i] < a[i+1] \| i=1..N-1; | A set of pairs <a[i], a[i+1]> is included into ρ. |

**Neighborhood Relation.** Binary relation called neighborhood relation is defined on the set of DFs. Two DFs are defined to be neighbor-related if it is recommended to keep them close to each other, for example, in the memory of the same PE. Usually this is done for DFs, which are the input variables of the same FC, and location of them in the same PE leads to reduction of the total communication overhead. Neighborhood relation is used by run-time subsystem in the process of dynamic workload balancing. If some workload has to be transferred from one PE to another, the run-time subsystem tries to minimize the number of neighbor-related DFs in different PEs after workload migration. Neighborhood relation provides the use of regularity of data and computation structures of numerical algorithms for the optimization of their execution.

**Execution Template.** To optimize the performance of the most time- and resource-consuming parts of the FP an execution template (ET) can be defined. The ET is an oriented graph, ET=<N,E>, where N is the set of ET nodes, and E is the set of ET oriented edges, $E=\{<n_1,n_2> | n_1,n_2 \in N\}$. The nodes N are execution units, connected with each other by edges, which transfer values from one node to another. A number of FCs can mapped to ET nodes. Consider an example (fig. 4). FCs $a_i$ are mapped to node *A*, $b_i$ – to node B and FC c is mapped to node C. Execution of the part of FP, mapped to the ET is organized as follows. When a node has a value on each of the incoming edges, it executes corresponding FC without any additional checks. After execution the output values are promoted via output edges to other nodes. In such a way, ET specifies inflexibly the scheme of FCs execution, which reduces the run-time subsystem overhead of choosing FC for execution.

**Profiling.** Profiling is a process of gathering a profile, which is the run-time information about an FP execution. This information includes real order of FCs execution, FCs execution times, PEs workload over time information, etc. The profile is used by run-time subsystem to optimize next executions of the FP. For example, if during the FP execution some PEs were idle because the value of some DF *x* was not calculated in time, then this information will be extracted from the profile, and during the next execution of the FP the FCs, which provide yielding the value of *x* will be assigned for execution earlier (if possible). It can be achieved, for example, by increasing the priority of these FCs. In such a way, each next execution of the FP will be done more efficiently (up to some limit, of course), if the FP is run on the same multicomputer and with the similar input data.

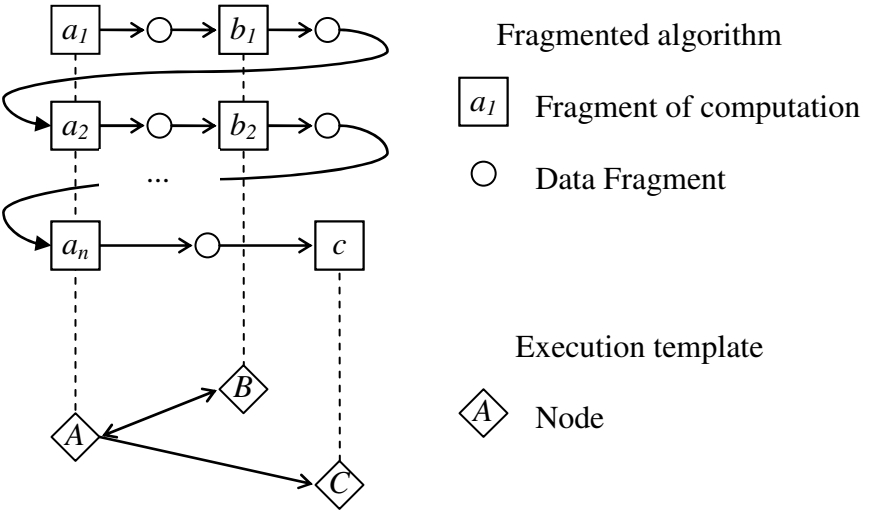



**Fig. 4.** FP to ET mapping

### 3.4 Testing

The performance of the LuNA run-time subsystem was tested on the implementation of a numerical model of self-gravitating stardust cloud using Particle-In-Cell method [18]. The parallel MPI-based implementation of the model was compared to the same implementation in LuNA programming system. The model was implemented as a fragmented program, using approach, described in [15]. The FP was executed by the LuNA run-time subsystem. The testing was performed on a cluster of the Siberian Supercomuter Center [19]. The testing results are shown in fig. 5.

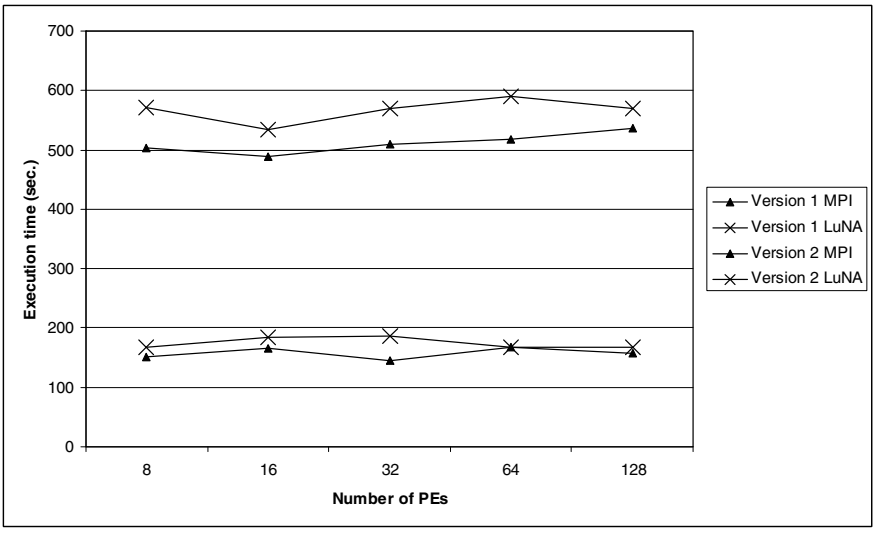



**Fig. 5.** Test results

There were two versions of the test, which differ by problem size. In the version 1 mesh size was 100×100×100 and the number of particles was $10^6$. In the version 2 mesh size was 200×200×200 and the number of particles was $10^7$. In both versions the problem size was increased with the PEs count increase (in the same proportion). In such a way, the execution time was approximate the same for the same version of the test.

As it is seen in the fig. 5 the difference in execution time between MPI and LuNA implementation is minor, which means, that the efficiency of hand-made MPI programs is reachable using LuNA programming system. However, to reach that efficiency, a FP has to be tuned up by the properly defined recommendations.

## 4 Conclusion

Taking into account the peculiarities of numerical algorithm provided low level of LuNA overhead, high performance of numerical algorithms implementation. Some additional LuNA modification are also planned to be implemented soon.

## References

1. Glushkov, V.M., Ignatiev, M.V., Myasnikov, V.A., Torgashev, V.A.: Recursive machines and computing technologies. In: Proceedings of the IFIP Congress, vol. 1, pp. 65–70. North-Holland Publish. Co., Amsterdam (1974)
2. Torgashev, V.A., Tsarev, I.V.: Programming facilities for organization of parallel computation in multicomputers of dynamic architecture. Programmirovanie (4), 53–67 (2001) (in Russian); (Sredstva organizatsii parallelnykh vychislenii i programmirovaniya v multiprocessorakh s dynamicheskoi architechturoi)
3. Valkovskii, V., Malyshkin, V.: Parallel Program Synthesis on the Basis of Computational Models. Novosibirsk, Nauka (1988) (in Russian) Sintez parallel'nykh program i system na vychislitel'nykh modelyakh)
4. Cell Superscalar, `http://www.bsc.es/cellsuperscalar`
5. Charm++, `http://charm.cs.uiuc.edu`
6. Shu, W., Kale, L.V.: Chare Kernel – a Runtime Support System for Parallel Computations. Journal of Parallel and Distributed Computing 11(3), 198–211 (1991)
7. Kalgin, K.V., Malyshkin, V.E., Nechaev, S.P., Tschukin, G.A.: Runtime System for Parallel Execution of Fragmented Subroutines. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 544–552. Springer, Heidelberg (2007)
8. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. ACM SIGPLAN Notices 30(8), 207–216 (1995)
9. Foster, I., Kesselman, C., Tuecke, S.: Nexus: Runtime Support for Task-Parallel Programming Languages. Cluster Computing 1(1), 95–107 (1998)
10. Chien, A.A., Karamcheti, V., Plevyak, J.: The Concert System – Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs. UIUC DCS Tech Report R-93-1815 (1993)
11. Grimshaw, A.S., Weissman, J.B., Strayer, W.T.: Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. ACM Transactions on Computer Systems (TOCS) 14(2), 139–170 (1996)

12. Benson, G.D., Olsson, R.A.: A Portable Run-Time System for the SR Concurrent Programming Language. In: Proceedings of the Workshop on Run-Time Systems for Parallel Programming, RTSPP (1997)
13. Dongarra, J.J., Sorensen, D.C., Hammarling, S.J.: Block reduction of matrices to condensed forms for eigenvalue computations. Journal of Computational and Applied Mathematics 27(1-2), 215–227 (1989); Special Issue on Parallel Algorithms for Numerical Linear Algebra
14. Malyshkin, V., Perepelkin, V.: Optimization of Parallel Execution of Numerical Programs in LuNA Fragmented Programming System. In: Hsu, C.-H., Malyshkin, V. (eds.) MTPP 2010. LNCS, vol. 6083, pp. 1–10. Springer, Heidelberg (2010)
15. Kraeva, M.A., Malyshkin, V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. The Int. Journal on Future Generation Computer Systems 17(6), 755–765 (2001)
16. Handbook of Research on Scalable Computing Technologies, p. 1021. IGI Global, USA (2010); ISBN 978-1-60566-661-7
17. Malyshkin, V.E., Sorokin, S.B., Chajuk, K.G.: Fragmentation of Numerical Algorithms for the Parallel Subroutines Library. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 331–343. Springer, Heidelberg (2009)
18. Kireev, S.E.: Parallel Implementation of the Particle-In-Cell Method for Modeling of Problems of Gravitation Cosmo-Dynamics. Parallelnaya Realizaciya Metoda Chastits v Yacheykah Dlya Modelirovaniya Zadach Gravitacionnoy Kosmodinamiki, Avtometriya 3, 32–39 (2006) (in Russian)
19. Siberian Supercomputer Center, `http://www.sscc.ru`

# Grid Computing for Sensitivity Analysis of Stochastic Biological Models

Ivan Merelli[1], Dario Pescini[2], Ettore Mosca[1], Paolo Cazzaniga[3], Carlo Maj[4], Giancarlo Mauri[4], and Luciano Milanesi[1]

[1] Istituto Tecnologie Biomediche
Consiglio Nazionale Ricerche
Via F.lli Cervi 93, 20090 Segrate, Italy
{ivan.merelli,ettore.mosca,luciano.milanesi}@itb.cnr.it
[2] Università degli Studi di Milano-Bicocca
Dipartimento di Statistica
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy
dario.pescini@unimib.it
[3] Università degli Studi di Bergamo
Dipartimento di Scienze della Persona
Piazzale S. Agostino 2, 24129 Bergamo, Italy
paolo.cazzaniga@unibg.it
[4] Università degli Studi di Milano-Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Viale Sarca 336, 20126 Milano, Italy
{carlo.maj,mauri}@disco.unimib.it

**Abstract.** Systems biology is a multidisciplinary research area aimed at investigating biological systems by developing mathematical models that approach the study and the analysis of both the structure and behaviour of a biological phenomenon from a system perspective. The dynamics described by such mathematical models can be deeply affected by many parameters, and an extensive exploration of the parameters space in order to find crucial factors is most of the time prohibitive since it requires the execution of a huge number of computer simulations. Sensitivity analysis techniques can help in understanding how much the uncertainty in the model outcome is determined by the uncertainties, or by the variations, of the model input factors (components, reactions and respective parameters). In this work we exploit the European Grid Infrastructure to manage the calculations required to perform the SA on a stochastic model of bacterial chemotaxis, using an improved version of the first order screening method of Morris. According to the results achieved in our exploratory analysis, the European Grid Infrastructure is a useful solution for distributing the stochastic simulations required to carry out the SA of a stochastic model. Considering that the more intensive the computation the more scalable the infrastructure, grid computing can be a suitable technology for large scale biological models analysis.

## 1 Introduction

Systems biology is a multidisciplinary research area aimed at investigating biological systems by developing mathematical models that approach the study and the analysis of

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 62–73, 2011.

both the structure and behaviour of a biological phenomenon from a system perspective. Recent experimental investigations at the single-cell level [1] have highlighted the presence of noise, due to the inherently stochastic interactions between molecular species occurring in low amounts inside the cell. Therefore, standard modelling approaches based on ordinary differential equations cannot effectively capture the effects of biological random processes, such as those that can lead the system to different states starting from the same initial conditions (e.g. lysis or lysogeny in phage-infected bacteria [2]). In recent years, many algorithms that perform stochastic simulations of biochemical reaction systems have proved their intrinsic suitability for reproducing the dynamics of many cellular processes [3].

Mechanistic mathematical models which represent real biological systems are usually composed of large numbers of components, which interact through many biochemical processes. In the analysis of such kind of systems, the dynamics can be deeply affected by many parameters, and an extensive exploration of the parameters space in order to find crucial factors is most of the time prohibitive since it requires the execution of a huge number of computer simulations. Moreover, in the field of stochastic simulations, several outcomes of the same parameter settings are needed to enquire statistical properties of the system dynamics.

There are several techniques devoted to the analysis of a model dynamics. For instance, *steady state analysis* concerns the identification of points in the space of reachable states where some properties of the system does not change over time (e.g. where the behaviour of the system is constant over time); *bifurcation analysis* studies the qualitative variation of the steady states (e.g. transition from oscillating to non oscillating regime) as a consequence of the variation of the parameters; *parameter sweep application* explores the parameters space of a system by means of independent experiments; *sensitivity analysis* relates the uncertainty of the input of a model (i.e. variations on parameters or initial conditions) to its output (namely, the resulting behaviour). Sensitivity analysis of biological models requires the execution of at least one simulation for each variation of an input variable. Moreover, all the different initial settings generated by modifying the input parameters lead to independent simulations of the system dynamics, and this makes a distributed architecture suitable for the execution of sensitivity analysis methods.

While in the state of the art some implementations of parameter sweep applications in the context of biological models analysis are available on grid infrastructure [4], to the best of our knowledge there are no sensitivity analysis tools developed on distributed platform. In this work we present a grid computing approach to sensitivity analysis of biological models, which is realised by distributing a large numbers of stochastic simulations performed by using tau-DPP [5] stochastic simulator on the European Grid Infrastructure (EGI). This infrastructure is the largest grid in Europe and the opportunity of using an effective production platform for our experiments is of great importance to critically analyse its performance by distributing the stochastic simulations aimed at the sensitivity analysis of biological models.

## 2 Sensitivity Analysis of Stochastic Models

In the mathematical modelling of biological systems, sensitivity analysis (SA) techniques can help in understanding how much the uncertainty in the model outcome is determined by the uncertainties, or by the variations of the model input factors (components, reactions and respective parameters). Moreover, the SA of the model output can also reveal which input factors bring about the most striking effects on the system behavior, and thus can be assumed to be good control points for its dynamics. Therefore, the knowledge on sensitive parameters can guide the design or facilitate the choice of which validation experiments are the most suitable to carry out, to reduce laboratory costs and efforts. Traditionally, SA has been diffusely applied to deterministic continuous models, by means of (derivative-based) local or global methods [6], though theories and tools for parametric sensitivity of discrete stochastic systems have recently been defined. In stochastic systems, these methods have to account for the inherent random effects over the simulations outcome, and demand the evaluation of the mean, or distribution/variance-based distances of many independent simulations [7,8].

### 2.1 Elementary Effects and Optimized Sampling of the Input Space

The method we used in this work relies on the calculation for each input factor of a number of incremental ratios, called Elementary Effects (EE) [9], from which basic statistics are computed to derive sensitivity information. We choose the EE method since it has been proven to be a very good compromise between accuracy and efficiency, especially for SA of large models.

In this method, we consider a model with $k$ input factors $X_i,\ i = 1, \ldots, k$ which varies in a $k$ dimensional unit cube across $p$ selected levels. This means that the input space is discretized into a $p$-level lattice $\Omega$. For a given value of the input factors vector $\boldsymbol{X}$, the elementary effect of the $i$th input factor is defined as:

$$EE_i = \frac{[Y(X_1, X_2, \ldots, X_{i-1}, X_i + \Delta, \ldots, X_k) - Y(X_1, X_2, ..., X_k)]}{\Delta} \qquad (1)$$

where $p$ is the number of levels and $\Delta$, the variation of the input factors, is a value in $\{1/(p-1), \ldots, 1 - 1/(p-1)\}$.

The distribution of elementary effects associated with the $i$th input factor is obtained by sampling different $\boldsymbol{X}$ from $\Omega$. In order to optimize the exploration of the input space we used here a refined sampling strategy proposed by Campolongo et al. [10]. By using this method, $r$ trajectories of $(k+1)$ points in the input space are generated, each trajectory providing $k$ elementary effects, one per input factor, for a total of $r(k+1)$ sample points. In order to create the trajectories, a base value $\boldsymbol{x^*}$ for the vector $\boldsymbol{X}$ is randomly selected in $\Omega$. $\boldsymbol{x^*}$ is not part of the trajectories but it is used to generate all the trajectory points, which are obtained starting from $\boldsymbol{x^*}$. The first trajectory point $\boldsymbol{x^1}$, is obtained by increasing one or more components of $\boldsymbol{x^*}$ by $\Delta$, so that $\boldsymbol{x^1}$ is still in $\Omega$. The second trajectory point $\boldsymbol{x^2}$, is generated from $\boldsymbol{x^*}$ with the requirement that it differs from $\boldsymbol{x^1}$ in the $i$th component, which has been either increased or decreased by $\Delta$, the index $i$ is randomly selected in the set $\{1, 2, \ldots, k\}$. The third sampling point $\boldsymbol{x^3}$, is generated from $\boldsymbol{x^*}$ with the property that $\boldsymbol{x^3}$ differs from $\boldsymbol{x^2}$ for only one component

$j$, for any $j \neq i$, where the component $j$ can be either increased or decreased by $\Delta$. The procedure continues following the same rules until $(k+1)$ points are generated. The trajectory produced with $(k+1)$ sampling points $\boldsymbol{x^1}$,$\boldsymbol{x^2}$,$\ldots$,$\boldsymbol{x^{k+1}}$ has the key property that two consecutive points differ in only one component and that any value of the base vector $\boldsymbol{x}^*$ has been selected at least once to be modified by $\Delta$.

Once we have created a pool of $N$ trajectories it is possible to select $r$ trajectories with the aim to maximize their spread in the input space. The concept of spread $D$ is based on the following definition of distance $d_{ml}$, between two trajectories $m$ and $l$:

$$d_{ml} = \begin{cases} \sum_{i=1}^{k+1} \sum_{j=1}^{k+1} \sqrt{\sum_{z=1}^{k} [X_z^{(i)}(m) - X_z^{(j)}(l)]} & \text{if } m \neq l \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $k$ is the number of the input factors and $x_z^i(m)$ indicates the $z$th coordinate of the $i$th point of the $m$th trajectory. The distance $d_{ml}$ represents the sum of the geometric distance between all the pairs of points of the two trajectories under analysis. Given this trajectory to trajectory distance, it is then possible to quantify the concept of spread $D$ in a set of $r$ trajectories as the squared sum of all the $d_{ml}$ distances generated by all the possible couples within the set. Stated in other words, we need to evaluate the distance $d_{ml}$ between all the possible $\frac{1}{2}N(N-1)$ couples of trajectories in the pool and then enumerate the $\binom{N}{r}$ possible subset of trajectories. So doing, it is possible to choose the best set of $r$ trajectories from the $N$ generated by maximizing $D$, in order to optimize the exploration of the input space.For instance, given three trajectories $a$, $b$ and $c$ (i.e. $r = 3$) the spread among them is defined as:

$$D_{abc} = \sqrt{d_{a,b}^2 + d_{a,c}^2 + d_{b,c}^2} \tag{3}$$

Following this strategy, we can select the combination of trajectories with the maximum spread in order to optimize the exploration of the input space.

### 2.2 Comparison of Stochastic Models Output: The Histogram Distance

In the context of stochastic modelling, each set of parameters generates qualitatively identical dynamics but quantitatively different. Indeed, to enquire statistical properties of the system we need to simulate a high number of times the system for each parametrisation; and if we need to compare the simulations results for the calculation of sensitivity measures, we have to cope with the comparison of distributions.

In this work, we used the histogram distance, which is computed as follows:

$$\delta(S,T) = \sum_{i=1}^{h} \left| \frac{\sum_{j=1}^{|S|} \chi(s_j, I_i)}{|S|} - \frac{\sum_{j=1}^{|T|} \chi(t_j, I_i)}{|T|} \right| \tag{4}$$

where $h$ is the number of histogram bins, $s_j \in S$ and $t_j \in T$ are model outputs (e.g. the number of molecules of a specific type at a given time instant), $|S|$ and $|T|$ are the

cardinalities of the multisets $S$ and $T$ and correspond to the number of simulations executed, the function $\chi$ returns 1 if the element $s_j$ belongs to the interval $I_i$, 0 otherwise. $I_i$ is the $i$th interval in the range, which runs from $m_{min} + \frac{(i-1)L}{h}$ to $m_{max} + \frac{iL}{h}$, where $m_{min} = min\{S \cup T\}$, $m_{max} = max\{S \cup T\}$ and $L = m_{max} - m_{min}$. With sufficient simulations runs, the use of the histogram distance is able to precisely describe the difference between the reference output of a model and another output (of the same model) obtained with one or more input factors varied [11].

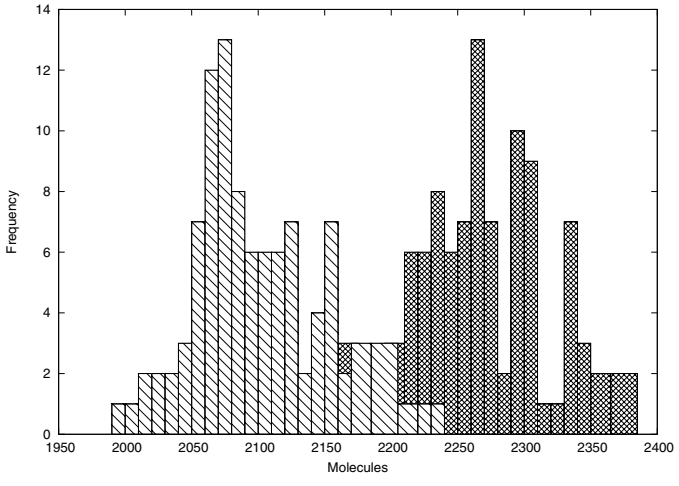



**Fig. 1.** Histogram plots obtained from 100 runs of the stochastic algorithm used to simulate the model with two different parametrisations taken from a trajectory obtained with the Morris sampling strategy

From a computational point of view, we need to perform many simulations with different model configurations in order to explore the input space factors and for each model configuration we need to repeat several simulations in order to calculate the histogram distribution of the output. In Fig. 1, an example of two histograms obtained from stochastic outcomes is plotted.

Given the fact that the simulations are independent, the task we want to perform is highly parallelizable and therefore the grid computing technology is a suitable approach to reduce the computational time needed to perform SA on large stochastic biological models.

## 3 The Bacterial Chemotaxis Case Study

As a case study, the SA was performed by using an improved version [10] of the first order screening method of Morris [9], on a stochastic model of bacterial chemotaxis. We choose this model because it represents a test of a realistic size compared to models currently available in systems biology literature: in fact, the bacterial chemotaxis model includes 59 parameters (i.e., the stochastic constants of the reactions) which represent the input factors in the context of SA.

From the biological point of view, chemotaxis is an efficient signal transduction pathway which allows bacterial cells to respond and adapt to their surrounding environment, by alternating random walks in homogeneous environments and longer directional running in presence of attractants or repellents [12,13]. A mechanistic model of the chemosensory system of *E. coli* bacteria, accounting for all protein-protein interactions and the feedback control mechanisms regulating the pathway, has been previously defined and used to analyze the temporal evolution of the pivotal chemotactic protein, CheYp, under different conditions [14].

## 4 The European Grid Infrastructure

In this work we exploit the EGI, a wide area grid platform for scientific applications composed of thousands of CPUs, which implements the Virtual Organisation (VO) paradigm [15]. The production framework is a large multi-science grid infrastructure, federating 250 resource centres worldwide, which provides comprehensively 20.000 CPUs and several Petabytes of storage. This infrastructure is used daily by thousands of scientists federated in over 200 VOs.

The EGI uses the gLite middleware [16], which was developed through the collaboration of a number of projects, like DataGrid, DataTag, Globus, GriPhyN, and LCG. The gLite software is an integrated set of tools designed to permit the sharing of computational resources and must be installed on a local server, defined as User Interface (UI), to allow the management of computations on the EGI. In particular, employing gLite, it is possible to submit grid jobs, monitor their state of advancement, and retrieve the output when the computations are successful or to resubmit them in case of failure. This grid infrastructure is highly scalable and allows computationally intensive challenges to be accomplished, but users must cope with the continuous dynamic reshape of the available resources, which is typical of loosely coupled distributed platforms.

To enable a secure connection to the remote resources, the grid middleware offers a well-established security system. The system relies on the Grid Security Infrastructure (GSI), which uses public key cryptography to recognise users. The access to remote clusters is granted by a Personal Certificate encoded in the X.509 format, which accompanies each job to authenticate the user. Moreover, users must be authorised to job submission by a VO, a grid community having similar tasks that vouches for them. In this test we joined the Biomed VO, which shares on average 2000 CPUs and welcomes applications in the bioinformatics field, in medical image processing, and more generally in biomedical data processing.

The resources available on the EGI are composed of a network of several Computing Elements (CEs), which are gateways for computer clusters where jobs are actually performed and an equal number of Storage Elements (SEs) that implement a distributed filesystem to store temporary files. The computational resources are connected to a Resource Broker (RB) that routes each job on a specific CE, taking into account the directives of the submission script, coded using the Job Description Language (JDL). In detail, the Workload Management System (WMS) is the RB service which schedules jobs by delivering them to the resource that best fits the requirements, balancing the computational load [17]. Although this brokering policy is not configurable by the user,

it provides high performance: bulk submission enables the sending of sets of independent jobs up to a rate of 50Hz for job submission and 0.5Hz for job dispatching to the CEs. Finally, each CE routes the incoming jobs to a batch queue system (PBS or LFS), which hides the farm of Working Nodes (WNs) where computations are effectively performed.

### 4.1 Performance Indexes

We will focus on two measures, hereby defined, to discuss the performances of EGI: the *crunching factor* and the overhead ratio. The crunching factor is a commonly used metric of the parallelisation gain achieved during a grid computation; it is defined as the ratio between the total expected CPU time over a single CPU and the duration of the grid computation, i.e. the time needed to accomplish the longest job:

$$c = \frac{nt}{\max(\tau_j)}, \quad j = 1, 2, \ldots, n \tag{5}$$

where $t$ is the expected time required for the computation of a single job using a single CPU, $\tau_j$ is the grid job time for job $j$ and $n$ is the number of grid jobs. Basically, the crunching factor $c$ represents the average number of CPUs used simultaneously along the whole computation, taking into account the longest job.

The second index is the *overhead ratio*, which is defined for a job $j$ as the ratio of the difference between the grid job time and the grid CPU time $\tau_j^{\text{cpu}}$ with the grid CPU time

$$o_j = \frac{\tau_j - \tau_j^{\text{cpu}}}{\tau_j^{\text{cpu}}} \tag{6}$$

The quantity $o_j$ is an indicator of the time spent "on the grid" with respect to the actual $\tau_j^{\text{cpu}}$.

## 5 Results

We performed the sensitivity analysis of the bacterial chemotaxis model presented in Section 3, in particular we computed the elementary effects by considering as input factors of the model, all the 59 stochastic constants associated to the biochemical reactions.

The trajectories generation process has been accomplished by using a lattice having $p = 4$ levels. Starting from a randomly generated point in the $k$ dimensional lattice (where $k = 59$ is the number of input factor), we generated $N = 40$ trajectories and then we computed all the distances between each pair of trajectories. The distances have been used to compute the spread of the $\binom{N}{r}$ possible subset of trajectories (where $r = 10$), and then the $r$ trajectories with the highest spread have been used to compute the the elementary effects of the input factors of the model.

The $r$ trajectories have been converted into $r(k+1)$ parametrisations for the stochastic simulations using ranges for stochastic constants which span two orders of magnitude above and below the reference values used to obtain a correct dynamics of the

system [14]. For each parametrisation, 100 simulations have been executed using the $\tau$-DPP algorithm in order to build the histograms used to compute the elementary effects; hence, a total of $100r(k+1) = 6 \cdot 10^4$ simulations were required to perform a run of SA.

We distributed over the EGI four runs of SA in which we modified the granularity of grid jobs, ranging from 60 to 6000 simulations per job. All the jobs were submitted to the grid through a User Interface, which periodically queries the RBs in order to retrieve the output in case jobs are correctly finished or to resubmit them in case of failure. Data about the time required to accomplish the whole computation are reported in Table 1, against the 13 days required when using a single CPU. The best crunching factor was 20.1 and was obtained during the run 2, in which we split the set of $6{\cdot}10^4$ simulations in 100 grid jobs, as reported in Table 1. The granularity used during run 2 resulted in grid job times ranging from 211.9min to 397.8min, Fig.2. Taking into account also the overhead ratio, reported in Fig. 3, it is evident that the best performance achieved in run 2 corresponded to the best granularity: in fact, a further increase of the number of jobs (run 1) yielded a lower percentage of job successfully completed at the first submission and a higher overhead ratio (which increases as the the grid job cpu time decreases), while the reduction of the job number (runs 3 and 4) determined better job success rate and better overhead ratio, but, due to the lower parallelism, the overall performance is worse.

**Table 1.** Setting and performances of the four runs of SA distributed over the EGI; $n$ is the number of jobs, success rate is the percentage of the jobs successfully finished at the first submission and $c$ is the crunching factor

| Run | n | Simulations / Job | Success rate [%] | max $(\tau_j)$ [min] | c |
|---|---|---|---|---|---|
| 1 | 1000 | 60 | 76 | 1171.0 | 17.1 |
| 2 | 100 | 600 | 84 | 994.6 | 20.1 |
| 3 | 20 | 3000 | 90 | 1742.5 | 11.5 |
| 4 | 10 | 6000 | 100 | 2646.5 | 7.6 |

As a matter of fact, once we obtained the results of the simulations of the $r(k+1)$ parametrisations, we built the histograms associated to each set of input factors and we used the histogram distance described in Section 2 to identify the input factor having the highest effect on the model dynamics. In particular, the histogram distance has been computed between the outcomes referred to adjacent points of each trajectory. So doing, we calculated $r$ elementary effects for each stochastic constant.

The elementary effects computed as histogram distances have been then used to evaluate the mean and standard deviation of the input factor effects. Following the method described in [10] we computed the values of $\mu^*$ and $\sigma$ of the 59 parameters of the bacterial chemotaxis model; in Fig. 4 we plot these results: the log-scale plot of $\mu^*$ and $\sigma$ (top left), the zoom in linear scale of the input factors with $\mu^*$ and $\sigma$ in the intarvals $[0, 1]$ (top right), $[1, 100]$ (bottom left) and $[100, 40000]$ (bottom right). For the sake of readability, we did not labelled the points of the graphs with the corresponding stochastic constant index. However, we reported in Table 2 the 10 most influential input factors of the model.

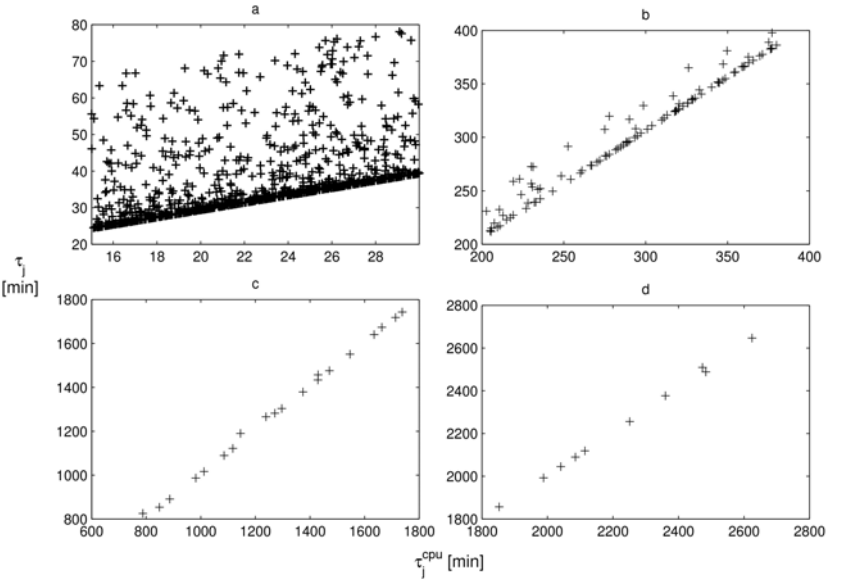



**Fig. 2.** Scatter plots of grid job times $\tau_j$ (vertical axes) and gid job cpu times (horizontal axes) $\tau_j^{\text{cpu}}$ for (a) run 1, (b) run 2, (c) run 3 and (d) run 4

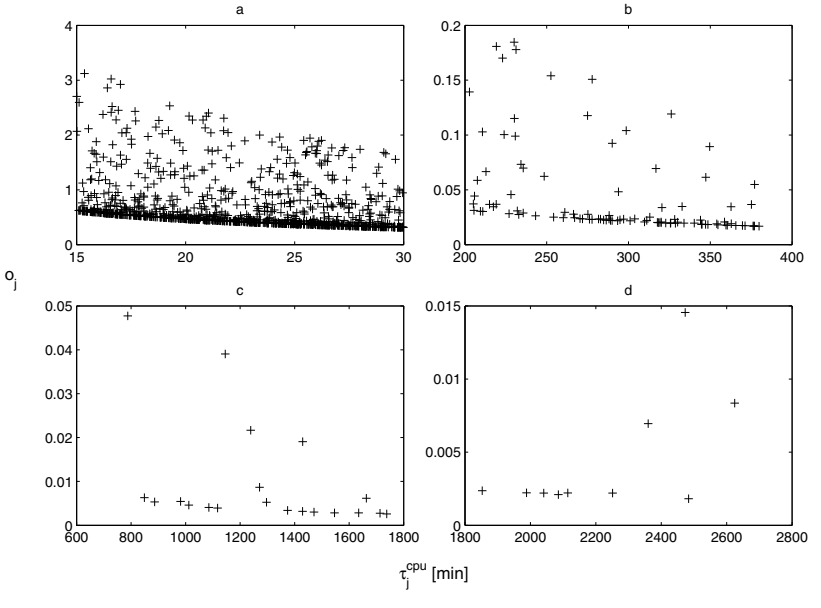



**Fig. 3.** Scatter plots of overhead ratio $o_j$ values (vertical axes) and grid job cpu times $\tau_j^{\text{cpu}}$ (horizontal axes) for (a) run 1, (b) run 2, (c) run 3 and (d) run 4

**Table 2.** $\mu^*$ and $\sigma$ values of the most influential input factors of the bacterial chemotaxis model

| **Reaction no.** | $\mu^*$ | $\sigma$ |
|---|---|---|
| 13 | 37204.7 | 24776.7 |
| 5 | 36061.1 | 27723.5 |
| 18 | 6231.71 | 7407.27 |
| 25 | 5758.29 | 7288.1 |
| 37 | 3637.65 | 2897.85 |
| 30 | 2649.92 | 2857.51 |
| 42 | 1803.36 | 1461.62 |
| 49 | 1175.78 | 1277.24 |
| 54 | 792.272 | 840.66 |
| 15 | 124.342 | 102.439 |

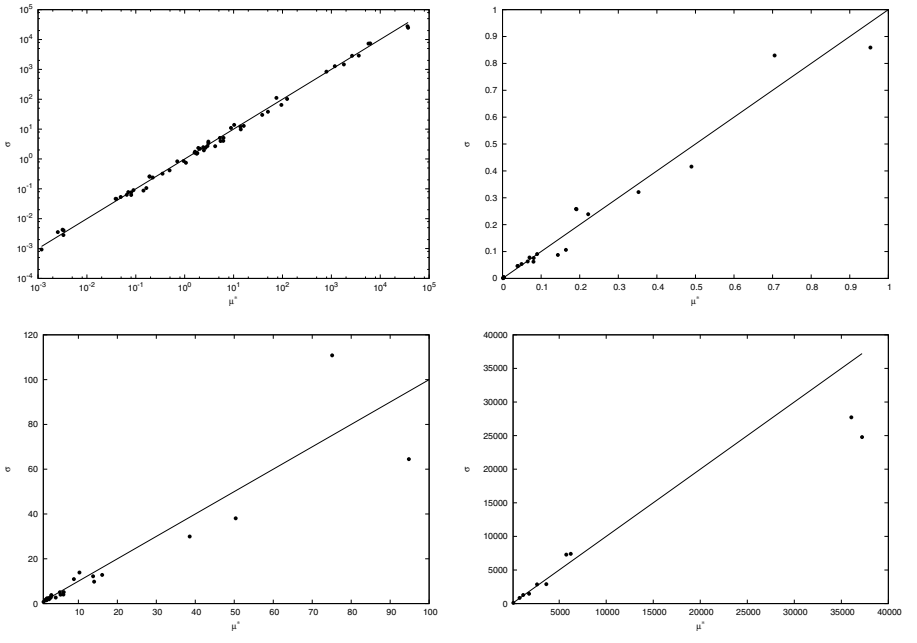


**Fig. 4.** Mean and standard deviation of the elementary effects of the bacterial chemotaxis model. Log-scale plot of the elementary effects (*top left*); zoom on the elementary effects with $\mu^*$ and $\sigma$ in the interval $[0, 1]$ (*top right*), $[1, 100]$ (*bottom left*) and $[100, 40000]$ (*bottom right*). In all graphs the elementary effects (points) are plotted along with the bisector.

The results of the sensitivity analysis of the bacterial chemotaxis model show that among the 10 most influential input factors, the stochastic constants of reactions 13, 5, 37, 42 and 15 have a linear effect since $\mu^*$ values are higher than $\sigma$, while the other constants have non-linear effects. Moreover, there are 20 stochastic constants whose $\mu^*$ and $\sigma$ values are smaller than one; hence, they have negligible effect on the model and can be discarded during a successive exhaustive analysis.

We do not give here an interpretation of the biological meaning of this analysis because in this paper we focus on the evaluation of the performance of the grid for the application of sensitivity analysis methods.

## 6 Conclusions

We performed an exploratory analysis to evaluate the reliability of the EGI in performing a large number of simulations of a stochastic model, which is requested by the EE approach to SA. More precisely, we highlighted critical factors, bottlenecks and scalability of this platform, focusing on the issues related to stochastic simulations.

A crucial factor in performing a grid computation is the identification of a suitable strategy for splitting it into a set of grid jobs, which means defining the granularity of the computation. The computation of long jobs on the grid may cause significant data loss in case of system failure or problems related to data transfer. On the other hand, the execution of a large number of short jobs raises the total latency time in the batch queues, affecting the global performance of the system (see Tab. 1 and Fig. 3).

Concerning the job granularity, once the optimal grid job computational time is found, the determination of the optimal number of simulations per job is not an easy task, because the model simulation time varies according to the values of the input factors. This is due to the fact that the dynamics is not uniformly sampled and that the number of points in the dynamics is affected both by the stochastic constants value and by the molecular amounts occurring in the system.

The strategy that we apply in order to increase the accuracy of the SA is by itself time consuming. It is possible to reveal the related computational effort, by reviewing the main steps of this strategy. First, a high number ($N$) of trajectories has to be generated. Then, the $r$ trajectories that have the greatest spread are selected, where the spread is measured as the sum of the $r$ trajectory-trajectory distances. This means that we first need to evaluate the distance $d_{ml}$ between all the possible $\frac{1}{2}N(N-1)$ couples of trajectories in the pool and then enumerate the $\binom{N}{r}$ possible subset of trajectories. To compute the spread of each of these subsets, a sum of $\frac{1}{2}r(r-1)$ terms is finally performed.

A key point in which the grid technology is a reliable approach to improve the accuracy of the SA, it is the possibility to increase the number of dynamics executions to compute the histogram distance. In this exploratory work, the number of simulations that we performed for each point of the trajectory can be considered a lower bound, as it can be seen in Fig. 1.

In conclusion, the EGI proved to be a useful solution for distributing the stochastic simulations required to carry out the SA of a stochastic model. This platform proved its efficiency in the context of our test and considering that the more intensive the computation the more scalable the infrastructure, grid computing can be a suitable technology for large scale biological models analysis.

**Acknowledgements.** This work has been supported by the italian projects CNR-BIOINFORMATICS, FIRB ITALBIONET (RBPR05ZK2Z) and Flagship Initiative INTEROMICS.

## References

1. Elowitz, M.B., Levine, A.J., Siggia, E.D., Swain, P.S.: Stochastic gene expression in a single cell. Science 297, 1183–1186 (2002)
2. Arkin, A., Ross, J., McAdams, H.H.: Stochastic kinetic analysis of developmental pathway bifurcation in phage $\lambda$-infected Escherichia coli cells. Genetics 149, 1633–1648 (1998)
3. Turner, T.E., Schnell, S., Burrage, K.: Stochastic approaches for modelling in vivo reactions. Comput. Biol. Chem. 28, 165–178 (2004)
4. Mosca, E., Merelli, I., Milanesi, L., Cazzaniga, P., Pescini, D., Mauri, G.: Stochastic simulations on a grid framework for parameter sweep applications in biological models. In: International Workshop on High Performance Computational Systems Biology, HIBI 2009, pp. 33–42 (2009)
5. Cazzaniga, P., Pescini, D., Besozzi, D., Mauri, G.: Tau leaping stochastic simulation method in P systems. In: Hoogeboom, H.J., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2006. LNCS, vol. 4361, pp. 298–313. Springer, Heidelberg (2006)
6. Saltelli, A., Ratto, M., Andres, T.: Global sensitivity analysis: the primer. Wiley Online Library (2008)
7. Gunawan, R., Cao, Y., Petzold, L., Doyle, F.J.: Sensitivity analysis of discrete stochastic systems. Biophys. J. 88, 2530–2540 (2005)
8. Plyasunov, S., Arkin, A.P.: Efficient stochastic sensitivity analysis of discrete event systems. J. Comp. Phys. 221, 724–738 (2007)
9. Morris, M.: Factorial sampling plans for preliminary computational experiments. Technometrics 33, 161–174 (1991)
10. Campolongo, F., Cariboni, J., Saltelli, A.: An effective screening design for sensitivity analysis of large models. Environmental modelling & software 22, 1509–1518 (2007)
11. Degasperi, A., Gilmore, S.: Sensitivity analysis of stochastic models of bistable biochemical reactions. In: Bernardo, M., Degano, P., Tennenholtz, M. (eds.) SFM 2008. LNCS, vol. 5016, pp. 1–20. Springer, Heidelberg (2008)
12. Jurica, M.S., Stoddard, B.L.: Mind your b's and r's: bacterial chemotaxis, signal transduction and protein recognition. Structure 6, 809–813 (1998)
13. Wadhams, G.H., Armitage, J.P.: Making sense of it all: bacterial chemotaxis. Nat. Rev. Mol. Cell Biol. 5, 1024–1037 (2004)
14. Besozzi, D., Cazzaniga, P., Dugo, M., Pescini, D., Mauri, G.: A study on the combined interplay between stochastic fluctuations and the number of flagella in bacterial chemotaxis. EPTCS 6, 47–62 (2009)
15. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: enabling scalable virtual organizations. Int. J. High Perform. Comput. Appl. 15, 200–222 (2001)
16. Laure, E., Fisher, S., Frohner, A., Grandi, C., Kunszt, P., Krenek, A., Mulmo, O., Pacini, F., Prelz, F., White, J., Barroso, M., Bunic, P., Hemmer, F., Meglio, A.D., Edlund, A.: Programming the grid with glite. Comp. Meth. Sci. Tech. 12(1), 33–45 (2006)
17. Campana, S., Rebatto, D., Sciabá, A.: Experience with the glite workload management system in atlas monte carlo production on lcg. J. Phys. Conf. Ser. 119 (2008)

# Looking for Efficient Implementations of Concurrent Objects

Achour Mostéfaoui[1] and Michel Raynal[1,2]

[1] IUF, Université de Rennes 1, France
[2] IRISA, Université de Rennes 1, France
{achour,raynal}@irisa.fr

**Abstract.** As introduced by Taubenfeld, a contention-sensitive implementation of a concurrent object is an implementation such that the overhead introduced by locking is eliminated in the common cases, i.e., when there is no contention or when the operations accessing concurrently the object are non-interfering. This paper, that can be considered as an introductory paper to this topic, presents a methodological construction of a contention-sensitive implementation of a concurrent stack. In a contention-free context a push or pop operation does not rest on a lock mechanism and needs only six accesses to the shared memory. In case of concurrency a single lock is required. Moreover, the implementation is starvation-free (any operation is eventually executed). The paper, that presents the algorithms in an incremental way, visits also a family of liveness conditions and important concurrency-related concepts such as the notion of an abortable object.

**Keywords:** Abortable object, Asynchronous shared memory system, Atomic register, Compare&Swap, Contention manager, Contention-sensitiveness, Deadlock-freedom, Linearizability, Liveness, Lock-freedom, Non-blocking, Obstruction-freedom, Progress condition, Starvation-freedom, Synchronization.

## 1 Introduction

### 1.1 Concurrent Objects

*From mastering sequential algorithms to mastering concurrency.* The study of algorithms lies at the core of informatics and participates in establishing it as a *science* with strong results on what can be computed (decidability) and what can be efficiently computed (complexity). It is consequently unanimously accepted by the community that any curriculum for undergraduate students has to include lectures on sequential algorithms. This allows the students not only to better master the basic concepts, mechanisms, techniques, difficulties and subtleties that underlie the design of algorithms, but also understand the deep nature of computer science and computer engineering.

A challenge is now to attain the same goal in the context of concurrency. A *concurrent object* is an object that can be concurrently accessed by several processes. As any object, a concurrent object is defined by a set of operations that processes can invoke to cooperate through this object. These operations are the only way to access the internal representation of the object (that remains otherwise invisible to processes). We

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 74–87, 2011.

are interested here in concurrent objects that have a *sequential specification* and supply processes with *total operations*. A total operation is an operation that always returns a result (e.g., instead of blocking the invoking process, a dequeue() operation on an empty queue returns it the value $empty$).

*Linearizability.* The most popular safety property associated with concurrent objects is called *linearizability* [10]. This consistency condition extends *atomicity* to all objects defined by a sequential specification on total operations. More precisely, an implementation of an object satisfies *linearizability* (and we say that the object implementation is *linearizable*) if the operation invocations issued by the processes appear (from an external observer point of view) as if they have been executed sequentially, each invocation appearing as being executed instantaneously at some point of the time line between its start event and its end event. Said differently, an implementation is linearizable if it could have been produced by a sequential execution.

An important property associated with linearizable object implementations is that they compose for free. This means that, if both of the implementation of an object $A$ and the implementation of an object $B$ (each taken independently) are linearizable, then these implementations without any modification constitute a linearizable implementation of the composite object $(A, B)$. (It is important to notice that, in contrast to linearizability, other consistency conditions such as sequential consistency [14] or serializability [2] cannot be composed for free.)

*Traditional lock-based shared memory synchronization.* One of the most popular way to obtain linearizable implementations of concurrent objects is to use locks. Associating a single lock with an object prevents several processes/threads from accessing it simultaneously. This approach is based on the classical notion of mutual exclusion [3, 18, 24]. Interestingly, locks can take different shapes according to the abstraction level at which they are considered. The most known example of locks is certainly the *semaphore* object [3], on top of which more friendly (i.e., high level) lock-based abstractions (such as monitors [12] or serializers [11]) can be built. This approach has proved its usefulness in providing simple lock-based solutions to basic paradigms of shared memory synchronization (such as the producer-consumer problem, or the readers-writers problem). One of the main difficulties when designing a lock-based solution lies in ensuring deadlock prevention, and more generally, provable liveness guarantees. Moreover, from an implementation point of view, lock implementations can be costly in terms of underlying shared memory accesses [19].

*Contention-sensitive objects.* The notion of *contention-sensitive implementation* of a concurrent object has been recently introduced [26]. The contention-sensitiveness property means that the overhead due to locking has to be eliminated when there is no concurrency or when the operations that concurrently access an object are not interfering (e.g., enqueuing and dequeuing on a non-empty queue). In these cases (absence of contention or interference), a contention-sensitive implementation has to ensure that an operation on the object completes in a small (possibly constant) number of steps and without locks. Resorting to locks has to be restricted to concurrent conflicting operations only.

The first paper (to our knowledge) that introduced contention-sensitiveness (without giving it a name) is [16] where is presented a mutual exclusion algorithm in which, in a contention-free context, a process has to execute only seven shared memory accesses to enter the critical section. When there is contention, the number of shared memory accesses depends on the number of processes and the actual concurrency pattern.

### 1.2 Content of the Paper

*Abortable objects.* An *abortable* concurrent object behaves like an ordinary object when accessed sequentially, but may abort operations when accessed concurrently (in that case the aborted operation has no effect and returns a default value denoted $\perp$). This definition is inspired from, but stronger than, the definition of abortable objects introduced in [1] (in that paper, an aborted operation returns also $\perp$, but may or not take effect and this is not known by the invoking process). The important point (in both definitions) is that the state of the object is never left inconsistent.

As far as we know, the notion of abortable objects has first been discussed in [13] where is presented an abortable mutual exclusion object. At any time while it is executing its entry code, a process can stop competing for the critical section and this halting has not to alter the liveness of the other critical section requests.

*Progress conditions.* While it always considers linearizability as the implicit safety condition, this paper considers three progress conditions for concurrent objects: obstruction-freedom, non-blocking and starvation-freedom.

The *obstruction-freedom* progress condition [8] states that an operation is required to terminate only if it executes in a concurrency-free context (i.e., when there is no operation invoked concurrently which is also called *solo* execution). Hence, an obstruction-free implementation of an object does not prevent concurrent operation invocations from never terminating. Let us notice that the notion of an abortable object is stronger than obstruction-freedom: while both ensure object consistency, they differ in the liveness they provide to users. More precisely, both guarantee operation termination in concurrency-free context, obstruction-freedom does not guarantee operation termination in case of concurrency. Differently, all operation invocations of an abortable object do terminate (possibly returning the value $\perp$ in case of concurrency). Hence, an implementation of an abortable object trivially satisfies the obstruction-freedom progress condition while the opposite is not true.

An implementation of a concurrent object is *non-blocking* if it is obstruction-free and additionally guarantees that, in presence of concurrency, at least one concurrent operation terminates. In a failure-free context, non-blocking is the same as *deadlock-freedom*. Finally, an implementation of a concurrent object is *starvation-free* if any operation invoked by a process terminates[1]. Hence, we have a hierarchy of progress conditions: obstruction-freedom is strictly weaker than non-blocking that in turn is strictly weaker than starvation-freedom. This hierarchy defines a family of qualities of service for liveness properties.

[1] In presence of process crashes, starvation-freedom becomes $t$-*resilience* where $t$ is the maximum number of process that may crash. Moreover, in a set of $n$ processes, *wait-freedom* [7] is $(n-1)$-resilience.

*Content and roadmap.* This paper investigates the contention-sensitive approach for the implementation of concurrent objects as advocated by Taubenfeld in [26]. To that end, it considers a simple concurrent object, namely a shared stask (let us remark that a lock-based starvation-free implementation of such an object is trivial). Three algorithms implementing such an object are presented. The first algorithm provides the processes with an abortable stack. As already said, this means that concurrent push and pop operations are allowed to abort (i.e., return $\perp$), while a push or pop operation executed in a concurrency-free context has to terminate and return a non-$\perp$ value. This algorithm does not use locks and is consequently *lock-free*. The second algorithm, which is also lock-free and provides the processes with a non-blocking shared stack is a simple extension of the previous one.

Considering an underlying abortable shared stack, the third algorithm provides the processes with a contention-sensitive shared stack. When an operation is executed in a concurrency-free context, this algorithm uses no lock and, whatever the number of processes and the size of the stack, it requires only seven shared memory accesses. This means that the algorithm is particularly efficient in contention-free patterns. It resorts to a lock only when there are concurrent operations. Moreover, this algorithm ensures the starvation-freedom progress condition.

The algorithms are built incrementally. This helps better understand the mechanisms that are used to go from an abortable shared object to a contention-sensitive implementation that satisfies the starvation-freedom progress condition. Interestingly, the mechanism employed to ensure starvation-freedom constitute a *contention manager* that can be used to solve other fairness-related problems.

The paper is made up of 5 sections. Section 2 presents the computation model. Then Section 3 presents an algorithm implementing an abortable stack object and its extension to obtain a non-blocking implementation of a stack [22]. Section 4 presents a contention-sensitive algorithm that implements a starvation-free stack. This algorithm is based on a mechanism introduced in [26]. Finally, Section 5 concludes the paper. Last but not least, it is important to say that the aim of this paper is to promote the notion of contention-sensitive implementation of a concurrent object as an efficient alternative to fully lock-based implementations. The interested reader will find more general developments on the contention-sensitive approach in [26].

## 2 Computation Model

### 2.1 System Model

*Asynchronous processes and communication model.* The system is made up of $n$ sequential processes denoted $p_1$, $p_2$, ..., $p_n$. The integer $i$ is the identity of $p_i$. Each process proceeds to its own speed, which means that the processes are asynchronous.

Processes communicate by accessing a shared memory that consists of *atomic* registers. The base operations on a register are read, write and Compare&Swap (see below). "Atomic" means that all operations on a register $R$ appear as if they have been executed sequentially, and if operation $op1$ terminates before operation $op2$ starts, then $op1$ appears before $op2$ in the sequence.

Atomicity and linearizability denote the same consistency condition. The word "atomicity" is usually employed for read/write registers [15] while the word "linearizability" is employed for objects built on top of registers or other objects [10].

*Notation.* Shared registers are denoted with uppercase letters. In contrast, variables that are local to a process are denoted with lowercase letters.

*Failure model.* It is assumed that both processes and atomic registers are reliable. This helps better understand how the algorithms work. They actually can cope with process crash failures. This is shortly discussed in Section 5.

### 2.2 Compare and Swap Operation

*Definition.* The Compare&Swap operation, that is on an atomic register $X$ is denoted $X.\mathsf{C\&S}(old, new)$. It is a conditional write that does atomically the following: if the current value of $X$ is $old$, it assigns $new$ to $X$ and returns $true$; otherwise, it returns $false$.

**primitive** $X.\mathsf{C\&S}(old, new)$:
  **if** $(X = old)$ **then** $X \leftarrow new$; $\mathsf{return}(true)$ **else** $\mathsf{return}(false)$ **end if**.

This base operation exists on some machines such as Motorola 680x0, Intel, Sun, IBM 370 and SPARC architectures. In some cases, the returned value is not a boolean, but the previous value of $X$.

*The ABA problem.* When using Compare&Swap, a process $p_i$ usually does the following. It first reads the atomic register $X$ (obtaining value $a$) and later wants to update $X$ to a new value $c$ only if $X$ has not been modified by another process since it has been read by $p_i$. Hence, $p_i$ invokes $X.\mathsf{C\&S}(a, c)$. Unfortunately, the fact that this invocation returns $true$ to $p_i$ does not allow it to conclude that $X$ has not been modified since the last time it read it. This is because between the read of $X$ and the invocation $X.\mathsf{C\&S}(a, c)$ issued by $p_i$, $X$ may have been updated twice, first by a process $p_j$ that has successfully invoked $X.\mathsf{C\&S}(a, b)$ and then by a process $p_k$ that has successfully invoked $X.\mathsf{C\&S}(b, a)$, thereby restoring the value $a$ into $X$. This is called the ABA problem.

This problem can be solved by associating a new (tag) sequence number with each value that is written. The atomic register $X$ is then composed of several fields such as $\langle v, sn\rangle$ where $v$ is the current value of $X$ and $sn$ its associated sequence number. When it reads $X$ a process $p_i$ obtains consequently the pair $\langle v, sn\rangle$. When later it wants to conditionally writes $v'$ into $X$, it invokes $X.\mathsf{C\&S}(\langle v, sn\rangle, \langle v', sn + 1\rangle)$. It is easy to see that the write succeeds only if $X$ has continuously been equal to $\langle v, sn\rangle$.

## 3 Implementing an Abortable Stack and a Non-blocking Stack

The algorithm described in Figure 1 implements an abortable stack. It is a simplified version of the non-blocking algorithm introduced in [22] (which is presented in Figure 2).

*Operations.* An abortable stack has two operations denoted here weak_push($v$) (where $v$ is the value to be added at the top of the stack) and weak_pop(). An operation always succeeds when executed in a contention-free context. In that case weak_push($v$) returns *done* if $v$ has been pushed on the stack and *full* if the stack is full; weak_pop() returns the value that was at the top of the stack (and suppresses it from the stack) or returns *empty* if the stack is empty. In the other cases, an operation may abort, in which case it returns $\perp$.

*Shared data structures.* The stack is implemented with an atomic register denoted $TOP$ and an array of $k+1$ atomic registers denoted $STACK[0..k]$.

- $TOP$ has three fields that contain an index (to address an entry of $STACK$), a value and a counter. It is initialized to $\langle 0, \perp, 0\rangle$.
- Each atomic register $STACK[x]$ has two fields: $STACK[x].val$ that contains a value, and $STACK[x].sn$ that contains a sequence number (used to prevent the ABA problem as far as $STACK[x]$ is concerned).
  The capacity of the stack is $k$ and for $1 \leq x \leq k$ the register $STACK[x]$ is initialized to $\langle \perp, 0\rangle$. $STACK[0]$ is a dummy entry initialized to $\langle \perp, -1\rangle$ that always contains the default value $\perp$.

The array $STACK$ is used to store the content of the stack, and the register $TOP$ is used to store the index and the value of the element at the top of the stack. The content of both $TOP$ and $STACK[x]$ is modified with the help of the Compare&Swap operation. This operation is used to prevent erroneous modifications of the stack internal presentation.

The implementation is *lazy* in the sense that a stack operation assigns its new value to $TOP$ and leave the corresponding modification of $STACK$ to the next stack operation. Hence, while on the one hand a stack operation is lazy, on the other hand it has to help terminate the previous stack operation.

*The operation* weak_push($v$) When a process $p_i$ invokes weak_push($v$), it first reads the content of $TOP$ (that contains the last non-aborted operation on the stack) and stores its three fields in its local variables $index$, $value$ and $seqnb$ (line 01).

Then, $p_i$ helps terminate the previous non-aborted stack operation (line 02). That operation (be it a successful weak_push() or a successful weak_pop() as we will see later) required to write $\langle value, seqnb\rangle$ into $STACK[index]$. To that end $p_i$ invokes $STACK[index]$.C&S.$(old, new)$ with the appropriate values $old$ and $new$ in order the write be executed only if not yet done (lines 15-16).

After its help (that was successful if not yet done by another stack operation) to move the content of $TOP$ into $STACK[index]$, $p_i$ returns *full* if the stack is full (line 03). If the stack is not full, it tries to modify $TOP$ to register its push operation. This operation has to succeed if no other process modified $TOP$ since it was read by $p_i$ at line 01. In that case, $TOP$ takes its new value and weak_push($v$) succeeds. Otherwise it aborts (lines 06-07).

The triple of values associated with this push_try($v$) and to be written in $TOP$ if successful, is computed at lines (lines 04-05). Process $p_i$ first computes the last sequence number $sn\_of\_next$ used in $STACK[index+1]$ and then defines the new triple, namely, $newtop = \langle index+1, v, sn\_of\_next+1\rangle$ to be written first in $TOP$ and later in $STACK[index+1]$ thanks to the help provided by the next stack operation (let us remember that $sn\_of\_next+1$ is used to prevent the ABA problem).

```
operation weak_push(v):
(01) (index, value, seqnb) ← TOP;
(02) help(index, value, seqnb);
(03) if (index = k) then return(full) end if;
(04) sn_of_next ← STACK[index + 1].sn;
(05) newtop ← ⟨index + 1, v, sn_of_next + 1⟩;
(06) if TOP.C&S(⟨index, value, seqnb⟩, newtop)
(07)       then return(done) else return(⊥) end if.

operation weak_pop():
(08) (index, value, seqnb) ← TOP;
(09) help(index, value, seqnb);
(10) if (index = 0) then return(empty) end if;
(11) belowtop ← STACK[index − 1];
(12) newtop ← ⟨index − 1, belowtop.val, belowtop.sn + 1⟩;
(13) if TOP.C&S(⟨index, value, seqnb⟩, newtop)
(14)       then return(value) else return(⊥) end if.

procedure help(index, value, seqnb):
(15) stacktop ← STACK[index].val;
(16) STACK[index].C&S(⟨stacktop, seqnb − 1⟩, ⟨value, seqnb⟩).
```

**Fig. 1.** An abortable stack [22]

*The operation* weak_pop(). The algorithm implementing this operation has exactly the same structure as the previous one and is nearly the same. Its explanation is consequently left to the reader.

*Linearization points of successful* weak_push() *and* weak_pop() *operations.* The operations that do not abort are linearizable, i.e., they can be totally ordered on the time line, each operation being associated with a single point of the time line that is after its start event and before its end event. More precisely, a non-aborted operation appears as if it has been atomically executed

- when it reads $TOP$ (at line 01 or 08) if it returns *full* or *empty* (at line 03 or 10),
- or at the time at which it successfully executes $TOP.\mathsf{C\&S}(-,-)$ (line 06 or 13 according to the operation).

*From an abortable stack to a non-blocking stack.* A very simple algorithm that builds a non-blocking stack on top of an abortable stack is described in Figure 2. It is easy to see that this algorithm satisfies the obstruction-freedom property: an operation executed in a contention-free context returns always a non-$\perp$ value. It is also easy to see that no operation aborts: instead of aborting, an operation can loop forever. The interested reader will find in [22] a proof that, whatever the contention pattern, at least one operation always terminates (i.e., the algorithm is non-blocking).

```
operation non_blocking_push(v):
    repeat res ← weak_push(v) until res ≠ ⊥ end repeat;
    return(res).

operation non_blocking_pop():
    repeat res ← weak_pop() until res ≠ ⊥ end repeat;
    return(res).
```

**Fig. 2.** A linearizable non-blocking stack

## 4 A Contention-Sensitive Implementation of Stack

Let us remember that the aim is here the design of a contention-sensitive that implements a starvation-free stack, which means that the algorithm (a) is allowed to use a lock only when there is contention, and (b) has to execute a small and constant-bounded number of shared memory accesses when there is no contention.

The stack provides the processes with the operations denoted strong_push($v$) and strong_pop(). As the implementation of the contention-sensitiveness property is independent of the fact that the stack operation is strong_push() or strong_pop(), we describe a generic algorithm denoted strong_push_or_pop($par$) where $par = v$ if the operation is strong_push($v$) and $par = \perp$ if the operation is strong_pop(). Moreover, in the text of the algorithm weak_push_or_pop($par$) stands for weak_push($v$) or weak_pop() according to the context.

### 4.1 Data Structures

The implementation of the contention-sensitiveness property is based on two atomic registers, an array of atomic registers and a lock.

- The lock, denoted $LOCK$, is accessed by the operations lock() and unlock(). It is used to ensure that a single process executes the part of code bracketed by $LOCK$.lock() and $LOCK$.unlock(). This lock is assumed to be deadlock-free but it is not required to be starvation-free (see the remark below).
- $CONTENTION$ is a boolean register (initialized to $false$) that is set to $true$ by a process when it executes the underlying weak_operation($par$) operation. This allows a process that starts executing an operation to know that there is contention.
- $FLAG[i]$ boolean, is a boolean (initialized to $false$) that process $p_i$ sets to $true$ when it wants to execute a stack operation and there is contention. In that way, $p_i$ allows the other processes to know it is competing for the lock. Process $p_i$ sets $FLAG[i]$ to $false$ when it has executed its base weak_operation($par$) operation.
- $TURN$ contains a process identity. $TURN = i$ means that process $p_i$ has priority to use the lock. Its initial value is any process identity. In order to ensure starvation freedom, the next value of $TURN$ is $(TURN \mod n) + 1$. Such a round-robin mechanism is used in several mutual exclusion algorithms such as [17,23].

*Remark.* If the lock is starvation-free (i.e., it ensures that any requesting process will obtain the lock) the algorithm can be simplified. More precisely, the array $FLAG[1..n]$ and the register $TURN$ become useless and consequently the lines 04-05 and 10-11 can be suppressed from algorithm. Those are actually shared variables and the associated statements that transform a deadlock-free lock into a starvation-free lock.

```
operation strong_push_or_pop(par):  % par = v for push() and ⊥ for pop() %
(01)  if (¬CONTENTION)
(02)    then res ← weak_push_or_pop(par); if (res ≠ ⊥) then return(res) end if
(03)  end if;
(04)* FLAG[i] ← true;
(05)* wait ((TURN = i) ∨ (¬FLAG[TURN]));
(06)* LOCK.lock();
(07)  CONTENTION ← true;
(08)  repeat res ← weak_push_or_pop(par) until res ≠ ⊥ end repeat;
(09)  CONTENTION ← false;
(10)* FLAG[i] ← false;
(11)* if (¬FLAG[TURN]) then TURN ← (TURN mod n) + 1 end if;
(12)* LOCK.unlock();
(13)  return(res).
```

**Fig. 3.** A linearizable contention-sensitive starvation-free stack (code for $p_i$)

### 4.2 The Algorithm

The algorithm is described in Figure 3. It is made of two parts. A lock-free part and a lock-based part. (The lock-free part is called *shortcut* in [26].)

In the first part (lines 01-03), the invoking process $p_i$ reads $CONTENTION$ and, if this boolean is false, invokes the underlying weak_operation() operation. As we have seen if there is no contention this invocation returns a non-$\perp$ value and $p_i$ terminates. The number of shared memory accesses is then 6 (5 within the successful weak_push_or_pop() + 1 for the read of $CONTENTION$).

If $CONTENTION$ is equal to $true$ or weak_push_or_pop() returns $\perp$, $p_i$ knows there is contention. In that case, $p_i$ enters the second part (lines 04-13) which is made up of two phases.

- In the first phase (lines 04-05), $p_i$ first sets $FLAG[i]$ to $true$ to inform the other processes that it is competing for the critical section protected by the lock. Then, $p_i$ waits until either it is the process that is currently given priority ($TURN = i$) or the process that is currently given priority (namely $p_{TURN}$) is not competing ($FLAG[TURN] = false$). When one of these predicates becomes true, $p_i$ invokes $LOCK$.lock().
- The second phase (lines 06-13) starts when $p_i$ has gained mutual exclusion and is consequently the only process executing the lines 07-12.

  Process $p_i$ executes then repeatedly weak_push_or_pop($par$) until a successful invocation (line 08). When, this occurs it resets $CONTENTION$ and $FLAG[i]$ to

*false*. Then if the process $p_{TURN}$ that is currently given priority is not competing ($FLAG[TURN] = false$), $p_i$ gives priority to $p_{(TURN \mod n)+1}$ (line 11) before releasing the lock and returning its (non-$\perp$) result.

It is important to notice that, due to asynchrony and the code of lines 01-03, while a process $p_i$ is repeatedly executing weak_push_or_pop() at line 08, other processes can be executing weak_push_or_pop() at line 02 (because they read *false* from $CONTENTION$) and the execution of weak_push_or_pop() by these processes can be successful. As we will see in the proof, this does not cause a problem because (a) the number of strong_push_or_pop() invocations concurrent with the one of $p_i$ is bounded and (b) the future invocations of strong_push_or_pop() will read *true* from $CONTENTION$) and will consequently enter the second part of the algorithm in which they cannot bypass $p_i$.

### 4.3 Proof

**Lemma 1.** *If a process $p_i$ returns from its* strong_push($v$) *or* strong_pop() *invocation, it returns a non-$\perp$ value.*

**Proof.** The proof follows immediately from the predicate $res \neq \perp$ tested at line 02 if $p_i$ returns at that line, or tested at line 08 if $p_i$ returns line 13. $\square_{Lemma\ 1}$

**Lemma 2.** *If a process $p_i$ eventually succeeds in locking, it eventually terminates its current* strong_push() *or* strong_pop() *operation.*

**Proof.** Let us assume that a process $p_i$ succeeds in locking at time $t_1$. There is a consequently a finite time $t_2 > t_1$ from which $CONTENTION$ is *true*.

It follows that all the processes that invoke strong_push($v$) or strong_pop() after time $t_2$ skip the lock-free part and start competing for the lock after it has been acquired by $p_i$. Hence these processes cannot prevent $p_i$ from terminating its operation.

It follows that at most $x$ processes, $0 \leq x \leq n-1$, can be executing weak_push() or weak_pop() at line 02 while $p_i$ is executing weak_push() or weak_pop() at line 08. Let $X$ the corresponding set of processes. If $X = \emptyset$, the execution by $p_i$ of weak_push() or weak_pop() is concurrency-free and the the lemma trivially follows. Hence, let us consider the case $X \neq \emptyset$, As we have seen in Section 3, the processes in $X$ eventually terminate their executions of weak_push() or weak_pop(). In the worst case, $p_i$ loops executing weak_push() or weak_pop() (at line 08) until all the processes in $X$ have returned from their current invocation of weak_push() or weak_pop() at line 02. Let $t_3$ be such a time instant. If $p_i$ has not returned from its weak_push() or weak_pop() operation with a non-$\perp$ value before $t_3$, it follows from the previous observation that its first invocation of weak_push() or weak_pop() after $t_3$ will return a non-$\perp$ value, and consequently $p_i$ eventually terminates. $\square_{Lemma\ 2}$

**Lemma 3.** *If, while executing a* strong_push($v$) *or* strong_pop() *operation, a process $p_i$ reads true from $CONTENTION$ at line 01 or obtains $res = \perp$ at line 02, it eventually obtains the lock.*

**Proof.** Let us consider a process $p_i$ that sets $FLAG[i]$ to *true* (line 04). Hence, $p_i$ is a process as defined by the lemma assumption. We consider three cases.

1. Process $p_i$ exits the loop of line 05 because $TURN = i$.
   Let us observe that, in this case, $TURN$ remains equal to $i$ until $p_i$ resets $FLAG[i]$ to $false$ (line 10) and increases $TURN$ to $(i \mod n) + 1$ (line 11).
   It follows from the previous observation that any process $p_j$ ($j \neq i$) that executes the loop of line 05 after $TURN$ has been set to $i$, loops until $p_i$ executes the lines 10-11. Let $Y$ be this (possibly empty) set of processes.
   Hence, at most $x$ processes, $0 \leq x \leq n - (|Y| + 1)$, can compete with $p_i$ for obtaining the lock. As the lock is deadlock-free, it follows that, in the worst case, each of these processes obtains the lock before $p_i$. After they have obtained (and released) the lock, $p_i$ is the only process requesting the lock and necessarily obtains it, which completes the proof of the lemma for this case.
2. Process $p_i$ exits the loop of line 05 because $TURN = k \neq i$ and $FLAG[k]$ is equal to $false$. We have to show that $p_i$ eventually obtains the lock.
   Let us assume by contradiction that $p_i$ never obtains the lock. In the worst case, all processes are competing with $p_i$ to obtain the lock. Let $p_j$ be the process that obtains the lock (as the lock is deadlock-free, such a process does exist). Due to Lemma 2, it follows that $p_j$ eventually releases the lock. If $FLAG[TURN] = false$, $p_j$ advances $TURN$ to its successor $p_\ell$ (line 11) along the oriented logical ring $j, j+1, \ldots, n, 1, \ldots$. We have then $TURN = \ell$. If $\ell \neq i$, the reasoning is repeated replacing $p_j$ by $p_\ell$ (let us observe that, due a reasoning similar to Item 1, $p_\ell$ eventually obtains the look). As no process is skipped when $TURN$ is advanced to its successor, it follows that $TURN$ progresses from process to process until w have $TURN = i$. When this occurs, all processes that execute line 05 are blocked at that line until $p_i$ executes $FLAG[i] \leftarrow false$ (line 10).
   It follows than, from then on, the number of processes competing with $p_i$ to obtain the lock is bounded. A reasoning similar to the used one in Item 1 shows that $p_i$ eventually obtains the lock, which contradicts the initial assumption and concludes the proof of the lemma for that case.
3. Process $p_i$ never exits the loop of line 05. We show that this case cannot occur.
   Let us assume by contradiction that $p_i$ loops forever at line 05. This means that each time it evaluates the predicate at line 05 we have $TURN \neq i \wedge FLAG[TURN]$. Let $TURN = k_1$ when read by $p_i$.
   According to Item 1 and Item 2, it follows that $p_{k_1}$ eventually exits the loop line at 05 because it finds $TURN \neq k_1$ (Item 1) or $FLAG[TURN]$ is false (Item 2) and consequently it eventually obtains the lock. Hence $p_{k_1}$ later executes $FLAG[k_1] \leftarrow false$ (line 10) and $TURN \leftarrow (k_1 \mod n) + 1$ (line 11). Let $k_2$ be that process identity. If $k_2 = i$, $p_i$ exits the loop. Hence, let us assume that $k_2 \neq i$. If $p_i$ reads $false$ from $FLAG[k_2]$ it stops looping and we are in one of the two previous items. If $p_i$ reads always $true$ from $FLAG[k_2]$, we are in the same case as previously, replacing $k_1$ by $k_2$. We consider then process $p_{k_3}$ such that $k_3 = (k_2 \mod n) + 1$. Etc.
   If follows from the fact that no process is skipped when $TURN$ is modified at line 10 that eventually $p_i$ either is such that $TURN = i$ or reads $false$ from $FLAG[k_x]$ for some process identity such that $TURN = k_x$. When this happens we are in the case described in Item 1 or Item 2.

□ $_{Lemma\ 3}$

**Theorem 1.** *Any invocation of* strong_push() *or* strong_pop() *operation returns a non-*$\perp$ *value, and all invocations are linearizable. Moreover, the algorithm is contention-sensitive: any* strong_push() *or* strong_pop() *operation invoked in a contention-free context is lock-free and accesses six times the shared memory.*

**Proof.** The fact any strong_push() or strong_pop() operation invoked in a contention-free context is lock-free and accesses six times the shared memory follows directly from the text of the algorithm.

The fact that no operation returns $\perp$ follows from Lemma 1.

All invocations of strong_push() or strong_pop() that return at line 02 trivially terminate. The fact that all other invocations of strong_push() or strong_pop() terminate follows from Lemma 2 and Lemma 3.

The linearization point of a strong_push() (resp., strong_pop()) operation is the linearization point of the last weak_push() (resp., weak_pop()) operation it has executed (as defined in Section 3). $\square_{Theorem\ 1}$

### 4.4 From a Non-blocking Lock to a Starvation-Free Lock

When considering Figure 3, let us call starvation_free_lock($i$) the code defined by the starred lines 04-06 and starvation_free_unlock($i$) the code defined by the starred lines 10-12. The reader can notice that these two operations construct a starvation-free lock from a non-blocking one. The interested reader will find similar constructions in [23, 26].

## 5 Concluding Remarks

*Process crashes and unreliable registers.* When describing the previous algorithms which implement a concurrent task, we have considered that the processes where asynchronous but reliable. The reader can easily verify that these algorithms still work despite process crashes if no process crashes while holding the lock.

We have also assumed that the registers are reliable. Techniques to extend these algorithms to cope with unreliable registers have been studied in several works (e.g., [6]).

*Contention managers.* Contention managers have recently become a hot research topic. The interested reader will find in [4, 25] techniques to extends obstruction-free or non-blocking algorithms to wait-free algorithms (wait-freedom is starvation-freedom in presence of any number of process crashes [7]). She will also find a failure detector-based approach to boost obstruction-freedom or non-blocking to wait-freedom in [5].

More generally, the interested reader will find developments on concurrent objects in [9, 20, 21, 24].

## References

1. Aguilera, M.K., Frolund, S., Hadzilacos, V., Horn, S.L., Toueg, S.: Abortable and Query-abortable Objects and their Implementations. In: Proc. 26th Int'l ACM Symposium on Principles of Distributed Computing (PODC 2007), pp. 23–32. ACM Press, New York (2007)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems, p. 370. Addison Wesley Publishing Company, Reading (1987)

3. Dijkstra, E.W.D.: Hierarchical Ordering of Sequential Processes. Acta Informatica 1(1), 115–138 (1971)
4. Fich, F.E., Luchangco, V., Moir, M., Shavit, N.N.: Obstruction-free algorithms can be practically wait-free. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 78–92. Springer, Heidelberg (2005)
5. Guerraoui, R., Kapalka, M., Kuznetsov, P.: The Weakest Failure Detectors to Boost Obstruction-freedom. Distributed Computing 20(6), 415–433 (2008)
6. Guerraoui, R., Raynal, M.: From Unreliable Objects to Reliable Objects: The Case of Atomic Registers and Consensus. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 47–61. Springer, Heidelberg (2007)
7. Herlihy, M.P.: Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
8. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free Synchronization: Double-ended Queues as an Example. In: Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS 2003), pp. 522–529. IEEE Press, New York (2003)
9. Herlihy, M.P., Shavit, N.: The Art of Multiprocessor Programming, p. 508. Morgan Kaufman Pub., San Francisco (2008)
10. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
11. Hewitt, C.E., Atkinson, R.R.: Specification and Proof Techniques for Serializers. IEEE Transactions on Software Engineering SE5(1), 1–21 (1979)
12. Hoare, C.A.R.: Monitors: an Operating System Structuring Concept. Communications of the ACM 17(10), 549–557 (1974)
13. Jayanti, P.: Adaptive and Efficient Abortable Mutual Exclusion. In: Proc. 22th Int'l ACM Symposium on Principles of Distributed Computing (PODC 2003), pp. 295–304. ACM Press, New York (2003)
14. Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers C28(9), 690–691 (1979)
15. Lamport, L.: On Interprocess Communication, Part 1: Basic formalism, Part II: Algorithms. Distributed Computing 1(2), 77–101 (1986)
16. Lamport, L.: A Fast Mutual Exclusion Algorithm. ACM Transactions on Computer Systems 5(1), 1–11 (1987)
17. Peterson, G.L.: Myths about Mutual Exclusion. Information Processing Letters 12(3), 115–116 (1981)
18. Raynal, M.: Algorithms for Mutual Exclusion, p. 107. The MIT Press, Cambridge (1986); ISBN 0-262-18119-3
19. Raynal, M.: Locks Considered Harmful: A Look at Non-traditional Synchronization. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 369–380. Springer, Heidelberg (2008)
20. Raynal, M.: Shared Memory Synchronization in Presence of Failures: an Exercise-based Introduction. In: IEEE Int'l Conference on Complex, Intelligent and Software Intensive Systems (CISIS 2009), pp. 9–18. IEEE Press, New York (2009)
21. Raynal, M.: On the Implementation of Concurrent Objects. Technical Report 1968, IRISA, Université de Rennes, France (2011); To appear in a Springer Verlag LNCS special issue dedicated to the 75th birthday of Brian Randell
22. Shafiei, N.: Non-blocking Array-Based Algorithms for Stacks and Queues. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 55–66. Springer, Heidelberg (2008)

23. Suzuki, I., Kasami, T.: A Distributed Mutual Exclusion Algorithm. ACM Transactions on Computer Systems 3(4), 344–349 (1985)
24. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming, p. 423. Pearson Prentice-Hall, London (2006); ISBN 0-131-97259-6
25. Taubenfeld, G.: Efficient Transformations of Obstruction-Free Algorithms into Non-blocking Algorithms. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 450–464. Springer, Heidelberg (2007)
26. Taubenfeld, G.: Contention-Sensitive Data Structures and Algorithms. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 157–171. Springer, Heidelberg (2009)

# Cache Efficiency and Scalability on Multi-core Architectures

Thomas Müller[1], Carsten Trinitis[1], and Jasmin Smajic[2]

[1] Lehrstuhl für Rechnertechnik und Rechnerorganisation,
Institut für Informatik,
Technische Universität München, Germany
{Thomas.Mueller,Carsten.Trinitis}@in.tum.de
[2] ABB Corporate Research Switzerland
Segelhof 1, Baden, Switzerland
Jasmin.Smajic@ch.abb.com

**Abstract.** Two electrical engineering applications from industry partners dealing with sparse matrices were analyzed regarding cache efficiency and scalability on modern multi core systems. Two different contemporary multi-core architectures have been investigated, namely Intel's Westmere and AMD's Magny-Cours. This paper can be regarded as a continuation of the investigations presented in [14] and [15].

In addition, the SuiteSparseQR library for efficiently computing QR factorizations of sparse matrices was evaluated regarding scalability and cache efficiency.

**Keywords:** sparse matrix, multi-core, thread-to-core assignment, cache efficiency, Amdahl's Law, SuiteSparseQR.

## 1 Introduction

In today's world, most computer systems, including those being used for numerical simulations, are based on multi- or many-core processors. This is due to the fact that in recent years microprocessor development has undergone fundamental changes.

Until few years ago, the straight way to speed up performance was to increase clock rates with every new generation, leading to faster program execution without any modifications required. However, power consumption increases proportionally with the clock rate. On the other hand, due to Moore's Law, the number of available transistors on a chip doubles every one and a half years.

Hence, due to the above-mentioned reasons, the only way to provide increasing compute power without increasing clock frequency is to provide parallelism on chip, i.e. by placing multiple processor cores on the same die – typically around 8–12 at the time of this writing. With an increasing number of cores memory hierarchies are becoming more complex and adaptations of the code might be necessary to obtain maximum performance.

A description of the investigated applications as well as the hardware and software environment and tools used for the evaluation is given in section 2.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 88–97, 2011.

Section 3 shows the observations regarding the scalability of the applications on different systems and how it is influenced by e.g. different distribution of threads to cores and internal data structure layouts. Finally, section 4 concludes and provides a forecast to future work.

## 2 Software and Hardware Environment

### 2.1 Investigated Applications

Two electrical engineering applications from industry partners dealing with sparse matrices were analyzed regarding cache efficiency and scalability on modern multi-core systems. Both applications are parallelized using OpenMP [1].

**Application 1.** The first application is completely written in FORTRAN and uses no external libraries at all. It is based on a tool which was developed several years ago, is continuously improved ever since and which is deployed in several real-life environments.

Recently, efforts were made to parallelize this tool to benefit from the increasing number of cores in modern processors. For a more detailed description refer to [13,14,15].

**Application 2.** The second application is written in C/C++ and uses the SuiteSparseQR [4] library to compute QR factorizations of sparse matrices.

SuiteSparseQR is part of the extensive SuiteSparse [3] package for sparse matrices, which is developed by Timothy A. Davis at the department of Computer & Information Science & Engineering at the University of Florida. It implements the multifrontal method [11,7,12] and then relies on BLAS/LAPACK libraries to do most of the compute intensive work, like e.g. Householder reflections.

SuiteSparseQR supports several different methods to compute a fill-reducing ordering of the original matrix to reduce undesired fill-in during QR factorization. It also supports rank-detection during factorization and, finally, QR factorization can be split into a *symbolic* and *numeric* part. As long as the nonzero structure of a sparse matrix is not changed, its symbolic factorization can be reused for several numeric factorizations of the sparse matrix with updated values.

However, because of the structure of the used sparse matrices and the internal algorithms, the built in parallelization of various BLAS/LAPACK libraries does not achieve a significant performance improvement when using multiple cores. Therefore, parallelization is instead done at a higher level by creating several smaller tasks. The order in which these tasks may be processed is modeled as a dependency tree and tasks are then processed in parallel according to that tree.

### 2.2 Investigated Hardware and System Libraries

The investigated hardware systems were

- **Westmere**: 2 x Intel X5670 (Gulftown; 6 cores; 2.93 GHz; 6 x 256 KB L2 cache; 12 MB L3 cache); 32 GB DDR3 RAM.

– **Magny-Cours**: 2 x AMD Opteron 6174 (Magny-Cours; 12 cores; 2.20 GHz; 12 x 512 KB L2 cache; 2x6 MB L3 cache); 64 GB DDR3 RAM.

The Intel Compiler 11.1 was used to compile all C/C++ and FORTRAN code on all systems. We also did some comparisons – especially on the AMD system – using the GCC, as the Intel Compiler is often criticized for deliberately creating slower code for AMD processors. However, our tests showed, that the code compiled by the Intel Compiler always performed better on all systems.

Both ACML (AMD Core Math Library) [5] by AMD as well as MKL (Math Kernel Library) [9] by Intel were used as BLAS/LAPACK libraries and compared against each other. As we will show in section 3.1, our evaluation showed clearly, that it is crucial to select the appropriate BLAS/LAPACK implementation to get optimal performance – particularly if one wants to scale to a large number of threads.

## 3 Results

In this section we detail our various results. We start with pure scalability experiments in 3.1 and highlight the influence of different pinnings/distributions of threads to processor cores in 3.2. The growing impact of mostly sequential pre– and post–processing phases is discussed in 3.3 and, finally, different internal data structure layouts of one of the applications are compared in 3.4.

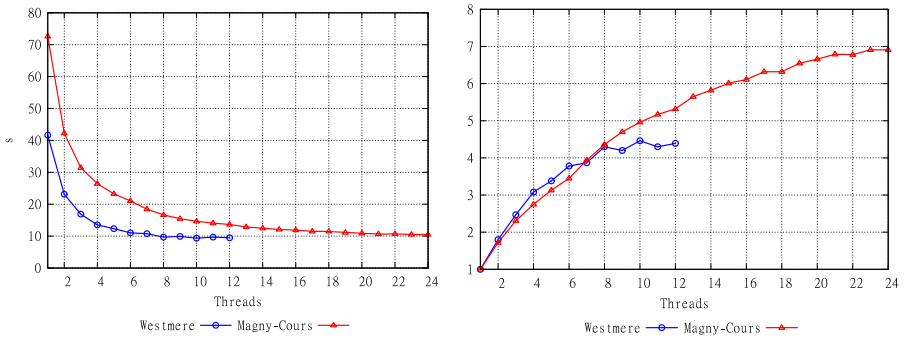
### 3.1 Scalability

At first, we executed both applications with an increasing number of threads on all systems to get get some simple scalability results and a first impression on how promising the parallelization of both applications is.

Figure 1 and 2 show the runtime and speedup of both applications on all evaluated systems. As one can easily see, the results differ noticeably on every system. The Westmere system is clearly faster, especially when using a smaller number of threads. This was to be expected, as the core clock frequency of 2.93 GHz of the Westmere processor is significantly higher than the 2.2 GHz of the Magny-Cours processor. On the other hand, starting from around 8–10 threads on the Magny-Cours speedup factors are considerably higher. However, even with all available 24 threads the total execution time on Magny-Cours is higher than with 12 threads on Westmere.

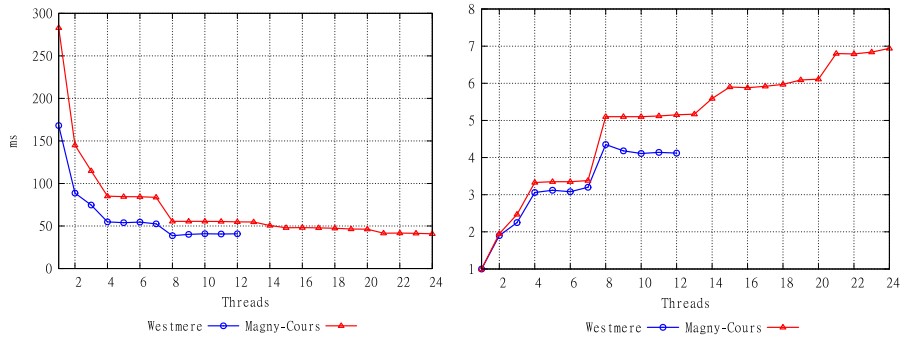
**BLAS/LAPACK Implementations.** Finally, figure 3 shows a comparison of the two different BLAS/LAPACK implementations used for the evaluation. Up to 6–8 threads performance is pretty much the same, with ACML having a small advantage. However, if the number of threads is further increased, ACML continues to improve performance while MKL pretty much stagnates.

Behavior on Westmere is similar, with MKL having a small advantage. However, the effect is more prominently visible on Magny-Cours, as there are more cores available for evaluation.




(a) Runtime in s (b) Speedup

**Fig. 1.** Runtime and speedup of application 1




(a) Runtime in ms (b) Speedup

**Fig. 2.** Runtime and speedup of application 2

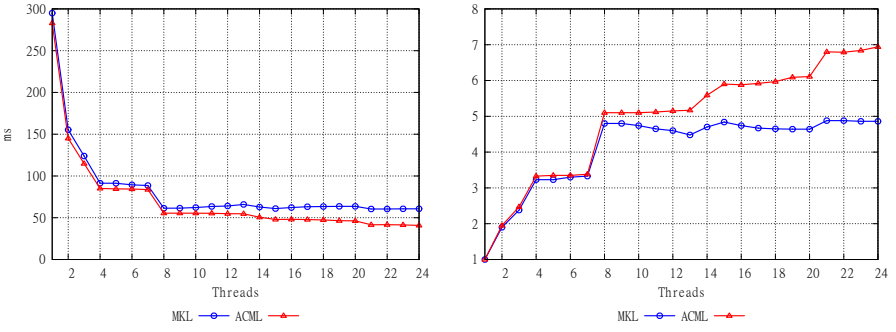
All in all, the overall speedup still leaves room for improvement, especially when using a very high number of threads. This is mostly due to the sequential pre– and post–processing phases of both applications and an imperfect load balancing within application 2, caused by an imbalance of the dependency tree. See section 3.3 for a more detailed analysis of the impact of the sequential phases.

### 3.2 Thread-To-Core Assignment Issues

We also evaluated the influence of different pinning strategies on overall performance. The first application for example achieves best performance on the Westmere system, when using only 10 of the available 12 cores. However, when using fewer cores than available, there are several possibilities to assign threads to cores and different assignments result in different performance, especially on multi-socket systems.

Two assignment or pinning strategies were evaluated:

- *compact*: all threads are placed as closely together as possible
- *scatter*: all threads are distributed as evenly on all packages as possible




(a) Runtime in ms (b) Speedup

**Fig. 3.** Comparison of runtime and speedup of application 2 on Magny-Cours with ACML and MKL

When for example running 8 threads on the Westmere system, the *compact* strategy would place 6 threads on the first package – thereby occupying every available core – and the remaining 2 threads on the second package. The *scatter* strategy on the other hand would place 4 threads on each of the 2 packages.

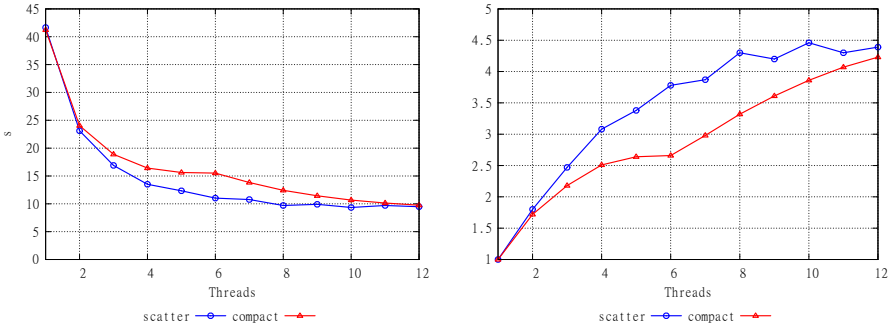
As one can see in figure 4 the strategy *scatter* results in noticeably better performance. Overall shortest runtime is achieved with 10 threads and the *scatter* strategy and is about 12% faster than 10 threads using the *compact* strategy and just over 2.5% faster than using 12 threads.

When using the *scatter* strategy, the available processor caches are pretty much evenly available to the running threads, i.e. every cache is accessed by the same number of threads. There is only very little communication between separate threads within application 1, so they don't need a cache which is shared among all threads. Instead, they benefits from an evenly distribution of threads as it is less likely that threads force eviction of another thread's data from the cache. Indeed, the number of total cache-misses for a complete run on the Westmere system dropped about 19%, while the variation of cache-misses for different runs with the same configuration is below 0.2%.

However, this result depends on the behavior of the application and the processor – especially the processor caches –, so it is not universally applicable. Application 2 for example shows no significant difference in performance when using different strategies – presumably because most work is done using BLAS/LAPACK libraries which are very carefully optimized and extremely cache-efficient.

### 3.3 Impact of Sequential Portions

As already mentioned in section 3.1, sequential pre– and post–processing phases limit the possible speedup of an application. This is a well known fact, often called Amdahl's Law [6], which is already more than 40 years old but still valid today.

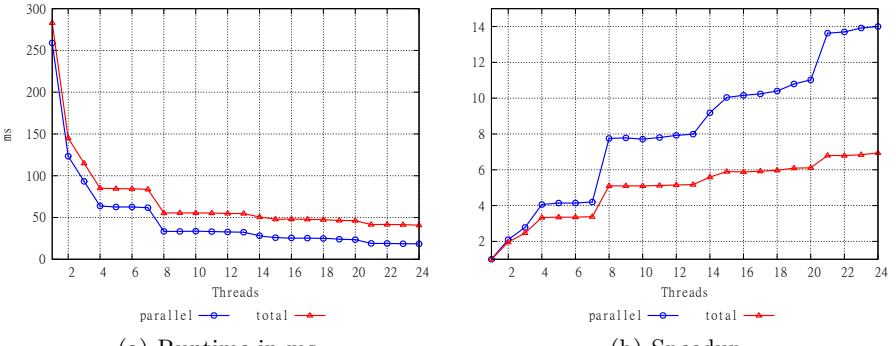



(a) Runtime in ms (b) Speedup

**Fig. 4.** Runtime and Speedup of application 1 with *compact* and *scatter* pinning strategy

Especially when parallelizing existing code which was developed, improved and tested over the course of several years, it is often hard to achieve a good parallelization with a very low sequential part without completely rewriting the application. However, a complete rewrite might take considerable time for coding and especially testing, which is usually hard to sell to business managers.

Yet, even newly developed code often has pre– and post–processing phases, which are caused by the underlying algorithms. Application 2 for example has such phases, which become very dominant when using a lot of threads, as can be seen in figure 5. Starting from 8 threads, total speedup only marginally increases, even though speedup within the parallelized section of code continues to improve noticeably. The sequential pre– and post–processing phases of application 2 account for only about 8% when running with just a single thread, but amount to about 54% when running with 24 threads.




(a) Runtime in ms (b) Speedup

**Fig. 5.** Comparison of parallel and total runtime and speedup of application 2 on Magny-Cours

### 3.4 Cache Behavior

Finally, an evaluation using performance counters was carried out for both applications to e.g. identify excessive cache misses, which can slow down an application significantly. To do this, we used the *perf* [2] tool, which is developed as part of the Linux kernel and uses the Linux Performance Counter subsystem that is part of newer Linux kernels.

For this evaluation we mainly concentrated on the event *Last Level Cache Misses* which is one of the *Pre-defined Architectural Performance Events* [8] with the following official definition: *This event counts each cache miss condition for references to the last level cache. The event count may include speculation, but excludes cache line fills due to hardware-prefetch.*

**Application 1.** An evaluation of application 1 using *perf* shows a summary of cache misses like the following:

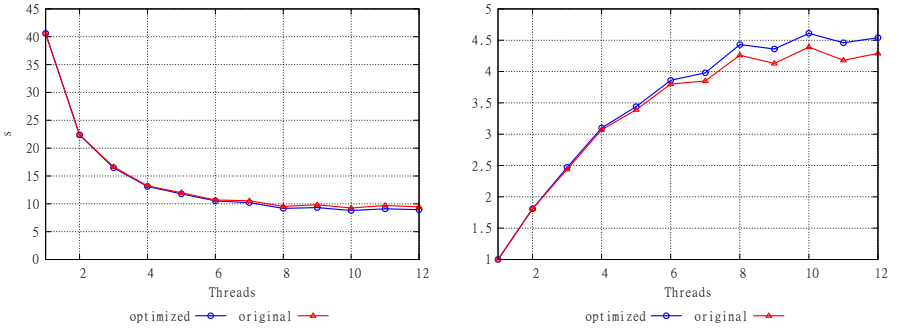
```
# Events: 8M cache-misses
#
# Overhead Samples   Cmd  Shared Object                    Symbol
# ........ ........  ...  ............  .......................
#
    16.20%  1317563  app  app           [.] wflowb_
    15.37%  1250298  app  app           [.] uregu1_
    10.47%   852031  app  app           [.] qflowb_
     6.34%   515889  app  app           [.] __intel_new_memset
     4.70%   382739  app  app           [.] rsmfus_
     4.31%   350538  app  app           [.] __intel_new_memcpy
     3.51%   285567  app  app           [.] qflow2_
     3.23%   262368  app  app           [.] rsmftu3_
     3.17%   257736  app  app           [.] bnout3_
```

As one can see, about 10–11% of cache-miss events are encountered during compiler generated memory initialization and copying (*__intel_new_memset* & *__intel_new_memcpy*). All other listed methods are part of the numerical algorithms within application 1 and often occur within varying loops.

The industry partner also provided a slightly modified version of the application, which effectively does the same work, but had its internal data structures partly redesigned. The intent of the modification was to increase memory access locality, which can increase processor cache efficiency, i.e. performance.

If a program accesses a value from memory, which is not already stored within the cache, the processor usually does not only load this single value, but instead fetches a continuous block of memory – a so-called cache-line –, which includes the value accessed by the program. If the program accesses two or more values located very closely together in memory, chances are good, that they are part of the same cache-line and only one memory access is necessary to retrieve the values. If those values are scattered within memory, more memory accesses might be necessary – one for each value in the worst case.

Indeed, our analysis has shown that the cache optimized version create about 6% less total cache-misses for a complete run on the Westmere system. The




(a) Runtime in s (b) Speedup

**Fig. 6.** Runtime and Speedup of application 1 with and without cache optimization

standard deviation of cache-misses between different runs of the same version is below 0.05%, so the improvement is really due to the modification of the internal data structures. This results in an improvement of wall-clock runtime of about 5–6%.

**Application 2.** An evaluation of application 2 using *perf* shows a summary of cache misses like the following:

```
# Events: 6M cache-misses
#
# Overhead Samples  Cmd  Shared Object                              Symbol
# ........ .......  ...  ..............  ...................................
#
    24.64% 1495865  app  app             [.] __intel_new_memset
    24.20% 1469025  app  ld-2.10.1       [.]             760de
    16.84% 1022093  app  app             [.] void spqr_stranspose2<double>()
    12.09%  733932  app  app             [.] void spqr_assemble<double>()
     6.39%  387624  app   7fd6be81e6b0   [.]     7fd6be81e6b0
     3.56%  216276  app  libmkl_lapack   [.] mkl_lapack_dlamch@plt
     2.05%  124681  app  app             [.] spqr_fsize()
     2.02%  122618  app  app             [.] void spqr_hpinv<double>()
     1.97%  119840  app  libmkl_mc3      [.] mkl_blas_dnrm2
     1.35%   82167  app  app             [.] __intel_new_memcpy
     0.89%   54241  app  app             [.] void spqr_kernel<double>()
     0.46%   27625  app  app             [.] long spqr_front<double>()
```

Again, compiler generated memory initialization and copying accounts for a noticeable amount of cache-miss events – this time for about 25%. However, the most interesting part are lines 2 and 5 with 24.20% and 6.39% respectively. Even though *perf* is unable to properly identify the method where these cache-miss events occurred, experiments strongly indicate, that this is due to compiler generated code for OpenMP and part of the synchronization mechanism. This suggests, that there are load balancing issues and threads often have to wait for results of other threads before they can continue working for themselves.

## 4 Conclusion and Future Work

In the previous section we have seen, that there are several obstacles to writing an efficient parallel application. One has to take care to keep sequential phases as small as possible – especially when trying to use a lot of threads. Proper load balancing across all used threads is also very important, as is the single core performance of the processor, as we have seen at the comparison of the Westmere and the Magny-Cours system. Internal data structures have to be carefully arranged to use processor caches as efficiently as possible.

And, finally, one has to resist the urge to always run an application with as many threads as cores are available. Instead, one has to carefully test the application with different threads and thread-to-core distributions, if optimal performance is to be achieved.

The last two – optimal number of threads and thread-to-core distribution – could be automated by tools like autopin [10], which is developed at Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München.

Future investigations will comprise extensive test runs on 256-bit AVX[1] based architectures including Intel's Sandy Bridge as well as AMD's new Bulldozer architecture. We also intend to investigate the feasibility of Intel's MIC[2] processor once it will become available.

## References

1. OpenMP: The OpenMP API specification for parallel programming, `http://www.openmp.org`
2. Perf Wiki, `https://perf.wiki.kernel.org`
3. SuiteSparse: a Suite of Sparse matrix packages, `http://www.cise.ufl.edu/research/sparse/SuiteSparse/`
4. SuiteSparseQR: multithreaded multifrontal sparse QR factorization, `http://www.cise.ufl.edu/research/sparse/SPQR/`
5. AMD: AMD Core Math Library, `http://www.amd.com/acml/`
6. Amdahl, G.: Validity of the single processor approach to achieving large-scale computing capabilities. In: AFIPS Conference Proceedings, vol. 30, pp. 483–485 (1967), `http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf`
7. Amestoy, P.R., Duff, I.S., Puglisi, C.: Multifrontal qr factorization in a multiprocessor environment. Numerical Linear Algebra with Applications 3(4), 275–300 (1996), `http://dx.doi.org/10.1002/SICI1099-150199607/083:4275::AID-NLA833.0.CO2-7`
8. Intel: Intel 64 and IA-32 Architectures Software Developers Manual; Volume 3B: System Programming Guide, Part 2, `http://www.intel.com/Assets/PDF/manual/253669.pdf`
9. Intel: Math Kernel Library, `http://software.intel.com/en-us/articles/intel-mkl/`

[1] Advanced-Vector-Extensions.
[2] Many-Integrated-Core.

10. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: autopin automated optimization of thread-to-core pinning on multicore systems. In: Stenström, P. (ed.) Transactions on High-Performance Embedded Architectures and Compilers III. LNCS, vol. 6590, pp. 219–235. Springer, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-19448-1_12
11. Liu, J.W.H.: The multifrontal method for sparse matrix solution: Theory and practice. SIAM Review 34(1), 82–109 (1992), http://link.aip.org/link/?SIR/34/82/1
12. Matstoms, P.: Sparse linear least squares problems in optimization. Computational Optimization and Applications 7, 89–110 (1997), http://dx.doi.org/10.1023/A:1008680131271
13. Tinney, W., Brandwajn, V., Chan, S.: Sparse vector methods. IEEE Transactions on Power Apparatus and Systems PAS 104(2), 295–301 (1985)
14. Trinitis, C., Küstner, T., Weidendorfer, J., Smajic, J.: Sparse matrix operations on multi-core architectures. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 41–48. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03275-2_5
15. Trinitis, C., Küstner, T., Weidendorfer, J., Smajic, J.: Sparse matrix operations on several multi-core architectures. The Journal of Supercomputing, 1–9 (2010), http://dx.doi.org/10.1007/s11227-010-0428-9

# Symbolic Algorithm for Generation Büchi Automata from LTL Formulas

Irina V. Shoshmina and Alexey B. Belyaev

Saint-Petersburg State Polytechnical University,
Distributed Computing and Networking Department, St. Petersburg, Russia
ishoshmina@dcn.ftk.spbstu.ru, belyaevab@gmail.com

**Abstract.** Model checking is a new techology developed to ensure the correctness of concurrent systems. In this paper we consider one of the algorithms included in this techology, an algorithm for constructing Büchi automaton from a given LTL formula. This algorithm uses an alternating automaton as an intermediate model while translating the LTL formula to a generalized Büchi automaton. We represent data structures and data manipulations with BDD to increase algorithm effectiveness. The algorithm is compared on time and resulting Büchi automaton size with well known LTL to Büchi realizations (SPIN, LTL2BA), and it shows its effectiveness for wide class of LTL formulas.

**Keywords:** concurrent system verification, model checking, LTL, BDD, alternating automaton, transition acceptance, Büchi automaton.

## 1 Introduction

The complexity of modern software constantly grows. As a consequence, the number of errors in programs grows too, especially in systems with parallel and distributed architecture.

It is well known that error detection in parallel, distributed, and multithreaded programs is not easy. Even when algorithms of each interacting process of parallel system are absolutely clear, it is difficult to understand the behavour of the entire system. While developing a parallel program a programmer should monitor the possible combinations of partially ordered events, which is much harder than to control completely ordered events in sequential programs. Parallel systems working correctly "almost always" may keep subtle errors over the years. Those errors may reveal in rare and critic situations. As a rule such errors cannot be found out by testing.

In recent years, a new approach to program verification, model checking, was developed. Model checking is a technology wich is the most effective for formal verification of parallel and distributed systems [7]. It provides a method to verify whether a given formula (usually a formula of temporal logic, in particular linear temporal logic — LTL) is true on a model of the system. A formula describes the desired requirements to the system behavior. Model checking algorithms carry

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 98–109, 2011.

out a full analasys of all possible system's runs. This method has great potential to increase the quality of distributed software systems.

One method of LTL model checking is based on a theoretical-automata approach. For this purpose, the LTL formula negation expressing a given system property is translated to a corespondent Büchi automaton. After that a parallel composition of this Büchi automaton and the system model represented by a Kripke structure is constructed. Our paper presents an effective algorithm of LTL formula to Büchi automaton translation.

## 2 Background

LTL formulas are built over a set $AP$ of atomic propositions, logical connectives and temporal operators U, X. All LTL formulas are formed according to the following grammar:

$$\varphi ::= p|\neg\varphi|\varphi \wedge \varphi|\mathrm{X}\varphi|\varphi\mathrm{U}\varphi$$

where $p \in AP$. Deliverable temporal operators are $\mathrm{F}\varphi = True\mathrm{U}\varphi$ and $\mathrm{G}\varphi = \neg\mathrm{F}(\neg\varphi)$.

In general, the complexity of constructing Büchi automaton grows exponentially on the length of the LTL formula. Such complexity is not an obstacle for checking small LTL formulas. But in practice, there are cases when the size of LTL formulas are enormous. In particular it is the case when the property $\varphi$ is verified according to some assumptions about the verified system behavior (so-called *fairness constraints*).

Here is an example of such property: "If infinitely often an alternator state will be consistent, than always by pressing the protection button the vessel power supply becomes active sometime in future". LTL formula which corresponds to this property is: $\mathrm{GF}p \rightarrow \mathrm{G}(q \rightarrow \mathrm{F}r)$, where $p$ stands for "the alternator is in the consistent state", $q$ stands for "the protection button is pressed", $r$ stands for "the vessel power supply is active". It is obvious that a state of a vessel power supply system depends on many parametres (such as pressure in the diesel engine, diesel engine temperature etc.). So such kind of constraints are fairly easily expressed in LTL by adding additional subformulas, but the size of the whole LTL formula for this property is growing:

$$(\mathrm{GF}p_1 \wedge \ldots \wedge \mathrm{GF}p_k) \rightarrow \varphi$$

In SPIN, which is specially designed for verification of parallel algorithms and protocols, generation of Büchi automaton from such LTL formula for $n = 6$ takes about one hour and a half. And for $n = 9$ the translation algorithm used in SPIN could not construct Büchi automaton at all.

The aim of this paper is to develop an effective algorithm for generation of Büchi automaton from an LTL formula. We use alternating automaton as intermediate model while translating the LTL formula to a generalized Büchi automaton. The main stages of our algorithm are (a) translation of the given LTL formula to an alternating automaton, (b) constructing a generalized Büchi

automaton with transition acceptance and (c) generation ordinary Büchi automaton from generalized one. We present states and transitions of all intermediate automata, as well as logical operations over them as Boolean functions using Binary Decision Diagrams. The algorithm is compared on time and resulting Büchi automaton size with well known LTL to Büchi realizations (SPIN, LTL2BA), and it shows its effectiveness for wide class of LTL formulas. The theoretical basis of using alternating automata for LTL formula representation is developed in [2,3].

## 3 Alternating Automata on Infinite Words

Unlike finite nondeterministic ordinary automata, alternating automata have existential and universal choices of a set of states in transition function. An existential choice (OR choice) implies a non-deterministic transition into one of the possible states. A universal choice (AND choice) means that a transition occurs simultaneously into all states corresponding to this choice.

**Definition 1.** *An alternating Büchi automaton is a tuple* $A = (Q, \Sigma, q^0, \delta, F)$, *where* $Q = \{q_0, q_1, \cdots, q_n\}$ *is a finite nonempty set of states,* $\Sigma$ *is a finite nonempty input alphabet,* $q^0 \in Q$ *is an initial state,* $F \subseteq Q$ *is a set of accepting states, and* $\delta : Q \times \Sigma \to L(Q)$ *is a transition function,* $L(Q)$ *is a free distributive lattice generated by* $Q$.

$L(Q)$ has two binary operations: $\wedge$ for universal choice, and $\vee$ for existential choice. The operations satisfy the usual laws: commutativity, associativity, distributivity, idempotency, and merging.

$L(Q)$ may be represented by a set of positive Boolean formulas over $S$ (without negation) $B^+(S)$ [1], if each state $q_i$ is accociated with a propositional variable $s_i$:

$$s_i = \begin{cases} 1 \text{ , } q_i \in \tilde{Q}, \text{ where } \tilde{Q} \text{ is a given subset of } Q \\ 0 \text{ , else} \end{cases} \tag{1}$$

Henceforth we use propositional variables $s$ instead of states $q$.

Positive Boolean formulas express universal and existential choices combinations unambiguously. If a transition $\delta(s, a)$ is nonempty then the automaton accepts $a$ being in the state $s$. Transition function $\delta(s, a) = s_1 \vee (s_2 \wedge s_3)$ means that being in the state $s$ the automaton accepts a word $aw$, if it accepts the word $w$ from the state $s_1$ or from both $s_2$ and $s_3$.

Existential and universal choices of alternating automata transitions has the only interpretation in the disjunctive normal form over positive Boolean formulas (PDNF).

The disjunctive normal form over positive Boolean formulas (PDNF) is used for the only interpretation of existential and universal choices of alternating automata. In PDNF a term $C$ is a conjunction of formulas $B^+(S) : C = \wedge_k s_k$, where no $s_k$ occurs more than once. Each element $e \in B^+(S)$ has a unique

representation in a disjunctive normal form (up to the order of terms), $e = \vee_i C_i$, where no term $C_i$ subsumes a $C_j$, $i \neq j$. It could be rewritten as $e = \vee_i \wedge_{k_i} s_{k_i}$.

A run of an alternating automaton is a tree rather than a sequence as it is for a nondeterministic Büchi automaton. Let $\beta$ is an infinite branch of a run $\sigma$. A set of states occurring infinitely often in the branch $\beta$ is $inf(\beta)$.

**Definition 2 (Büchi acceptance condition).** *An infinite branch $\beta$ of a run $\sigma$ is accepting, if $inf(\beta) \cap F \neq 0$. A run $\sigma$ accepts an infinite word $w$, if any its infinite branch is accepting.*

## 4 Symbolic Realization of LTL to Büchi Automaton Translation

The main idea of the symbolic approach to algorithms processing finite data structures consists in using Boolean characteristic functions representing finite sets. A characteristic function defined on a subset $\tilde{A} \subseteq A$ of a finite set $A$ is a Boolean function which indicates membership of an element in a subset $\tilde{A}$. All operations over finite sets are corresponding to Boolean operations over characteristic functions.

In our algorithm subsets of automaton states, transition functions, transition labels, labels of accepting states are all specified by Boolean characteristic functions. Alternating automaton definition is suitable for symbolic apporach. Alternating automaton states are encoded according to (1). Transition functions describing state sets in which transitions are carried out are presented by Boolean functions. Symbolic algorithms of main LTL to Büchi translation stages are stated in the next sections in detail.

In our algorithm we use Binary Decision Diagrams (BDD) as an effective form of Boolean functions representation. BDD is a directed acyclic binary graph without redundancy in its structure. More details about BDDs may be found, for example, in [7].

## 5 Generation an Alternating Automaton from LTL Formula

We construct an alternating automaton Büchi with input alphabet $2^{AP}$ for a given LTL formula $\varphi$ over a set $AP$ of atomic propositions, which accepts exactly all infinite words satisfying the formula and only them. Our algorithm is based on the theoretical background given in [2].

**Theorem 1.** *[2]. Given a LTL formula $\varphi$, one can build an alternating Büchi automaton $A_\varphi = (S, \Sigma, s_0, \delta, F)$, where $\Sigma = 2^{AP}$ and $|S|$ is in $O(|\varphi|)$, such that $L_\omega(A_\varphi)$ is exactly the set of computations satisfying the formula $\varphi$.*

All states of an alternating automaton $A_\varphi$ are labeled by subformulas of the given LTL formula $\varphi$. Next states are obtained supplying transition rules recursively for every state. Transitions are labeled with elements of $2^{AP}$.

As far as alternating automaton transitions are defined in positive Boolean functions, we transform a LTL formula into a canonical form, so-called *negation normal form* (NNF), where all negations are adjacent to atomic propositions: $\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$, $\neg(\mathrm{X}\varphi) = \mathrm{X}\neg\varphi$, $\neg(\varphi \mathrm{U} \psi) = \neg\varphi \mathrm{R} \neg\psi$. The grammar of LTL formulas in NNF is the following:

$$\varphi ::= p | \neg p | \varphi \vee \varphi | \varphi \wedge \varphi | \mathrm{X}\varphi | \varphi \mathrm{U} \varphi | \varphi \mathrm{R} \varphi$$

During syntactic analysis any subformula $\varphi_i$ of the temporal formula $\varphi$ is corresponded to a $A_\varphi$ state $s_i$. So there is a function $\tau : \varphi \to S$ labeling states with subformulas such that $\tau(\varphi_i) = s_i$.

*Example 1.* Let's consider a part of the property given above: $\varphi = \mathrm{G}(q \to \mathrm{F}r)$. The NNF of this formula is the following: $\varphi = false\ \mathrm{R}\ (\neg q \vee (true\ \mathrm{U}\ r))$. As the result of syntactic analysis the following subformulas would be extracted: $\varphi_1 = r$, $\varphi_2 = true \mathrm{U} \varphi_1$, $\varphi_3 = \neg q$, $\varphi_4 = \varphi_3 \vee \varphi_2$, $\varphi_5 = \varphi = false \mathrm{R} \varphi_4$, and constructed propositional variables $s_1, \ldots, s_5$ corresponding to states of an alternating automaton $B_\varphi$.

Let $\Sigma = 2^{AP}$. Transitions rules of automaton $A_\varphi$ are defined for logical connectives and temporal operators. For example, a state corresponding to a proposition $p \in AP$ accepts a symbol $a \in \Sigma$ if $p$ is included in this set:

$$\delta\big(\tau(p), a\big) = true, \text{if } p \in a \tag{2}$$

$$\delta\big(\tau(p), a\big) = false, \text{if } p \notin a \tag{3}$$

The transiton from a state with a proposition negation is defined similarly.

A transition function for a state labeled with a disjunction of formulas $\varphi$ and $\psi$ define a set of states in which transitions on $a$ from $\varphi$ or from $\psi$ are defined:

$$\delta\big(\tau(\varphi \vee \psi), a\big) = \delta\big(\tau(\varphi), a\big) \vee \delta\big(\tau(\psi), a\big) \tag{4}$$

A state corresponding to a temporal formula $\varphi \mathrm{U} \psi$ accepts $a$ if $a$ is accepted by a set of states in which a transition from $\psi$ exists or, otherwise, a transition from a set of states $\varphi$ and $\varphi \mathrm{U} \psi$ exists, because in this case Until obligation is not realized:

$$\delta\big(\tau(\varphi \mathrm{U} \psi), a\big) = \delta\big(\tau(\psi), a\big) \vee \big(\delta\big(\tau(\varphi), a\big) \wedge \tau(\varphi \mathrm{U} \psi)\big) \tag{5}$$

This definition corresponds to a recursive formula for *Until*: $\varphi \mathrm{U} \psi = \psi \vee \varphi \wedge X(\varphi \mathrm{U} \psi)$.

Similarly, transition functions for $\varphi \wedge \psi$, $\varphi \mathrm{R} \psi$, $\mathrm{X}\varphi$ are defined as follows:

$$\delta\big(\tau(\varphi \wedge \psi), a\big) = \delta\big(\tau(\varphi), a\big) \wedge \delta\big(\tau(\psi), a\big) \tag{6}$$

$$\delta\big(\tau(\varphi \mathrm{R} \psi), a\big) = \delta\big(\tau(\psi), a\big) \wedge (\delta\big(\tau(\varphi), a\big) \vee \tau(\varphi \mathrm{R} \psi)) \tag{7}$$

$$\delta\big(\tau(\mathrm{X}\varphi), a\big) = \tau(\varphi) \tag{8}$$

Joining all transitions from a state $s$ over $\Sigma$ we get a function $\delta(s) = \vee_{a \in \Sigma}$ $\delta(s, a)$. If $s = \tau(p)$ then $\delta(s) = \delta(\tau(p)) = p$. The change of rules (4-8) is obvious,

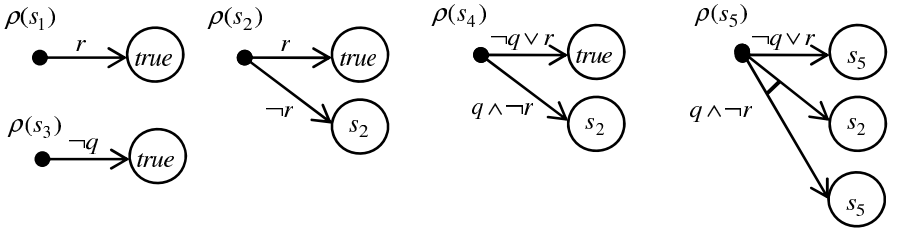



**Fig. 1.** Transition functions for $B_\varphi$ constructed from $\mathrm{G}(q \rightarrow \mathrm{F}r)$

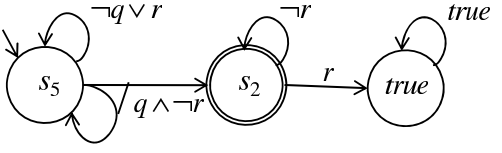



**Fig. 2.** The alternating automaton of LTL formula $\mathrm{G}(q \rightarrow \mathrm{F}r)$

for example, $\delta\big(\tau(\varphi \mathrm{U} \psi)\big) = \delta\big(\tau(\psi)\big) \vee \big(\delta\big(\tau(\varphi)\big) \wedge \tau(\varphi \mathrm{U} \psi)\big)$. Transition functions obtained from the rules (2-8) for $B_\varphi$ states (Example 1) are given on the Fig.1.

As a result of specifying the transition functions described above we construct a very weak alternating automata (VWAA) [6]. This VWAA contains states labeled by those subformulas of $\varphi$ which includes temporal operators, and those which labeled with *true*, and $\varphi$.

**Definition 3.** *An alternating automaton is called very weak alternating automaton, if a requirement of the partial order on $S$ is added to the Def. 7.*

As a consequence of Def. 3 there are no cycles formed by transitions between different states in VWAA.

**Definition 4 (co-Büchi acceptance condition).** *A run of $A_\varphi$ $\sigma$ is accepting if any infinite branch in $\sigma$ has only a finite number of nodes labeled by states from $F$ (in terms of Def. 2 if $inf(\beta) \cap F = 0$).*

For VWAA Büchi and co-Büchi acceptance conditions are equivalent up to redefinition of $F$. The final states for the co-Büchi condition are labeled with formulas with the temporal operator *Until*. The resulting VWAA $B_\varphi$ (Example 1) is shown on the Fig. 2, where the accepting co-Büchi state is labeled by a bold line.

## 6 From Alternating Automata to Generalized Büchi Automata with Transition-Based Acceptance

The next step of the algorithm is constructing a generalized Büchi automaton from the derived alternating one.

**Definition 5.** *A generalized Büchi automaton is a tuple* $GBA = (Z, \Sigma, z^0, \delta', T)$, *where:*

- $Z$ *is a finite set of states,*
- $\Sigma$ *is an finite alphabet,*
- $z^0 \in Z$ *is an initial state,*
- $\delta' : Z \to \Sigma \times Z$ *is a transition function,*
- $T = \{T_1, \cdots, T_r\}$ *is a set of accepting transitions.*

At this step we developed a symbolic algorithm based on the algorithm proposed and proved in [3]. Construction of GBA from an alternating automaton is similar to the classical algorithm for constructing a finite deterministic automaton from a finite nondeterministic one.

Each GBA state $z$ is a product of VWAA states, so $Z \subseteq 2^S$. Its transition function is a product of corresponding VWAA transitions. The initial state of the GBA coincides with the initial VWAA state: $z^0 = s^0$.

Since $F$ is the set of VWAA co-Büchi accepting states then a set of GBA accepting transitions groups are defined as following:

$$T = \{T_k \mid f_k \in F, 1 \leq k \leq r\}, \tag{9}$$

where

$$\begin{aligned} T_k = \{(z, a, z') \in Z \times \Sigma \times Z \mid f_k \notin z' \\ \vee \exists (b, z'') \in \delta(f_k) : (a \subseteq b \wedge f_k \notin z'' \wedge z'' \subseteq z')\}. \end{aligned} \tag{10}$$

The symbolic algorithm constructing the set $\Delta$ of GBA transitions is presented on Fig. 3. Consider some algorithm steps in detail.

Products of VWAA states forming GBA states are obtained from unique conjunctions entering into the disjunctive normal form of transition functions, starting from the initial state. To obtain these conjunctions from ordinary Boolean functions we add few operations.

A solution of a Boolean function $\delta(s)$ is a vector:

$$\xi = \big(\xi(p_1), \ldots, \xi(p_m); \xi(s_1), \ldots, \xi(s_n)\big),$$

where $\xi(x)$ is an assignment of a Boolean variable $x$, $p_i \in AP$, $s_i \in S$.

Consider a set of solutions $\zeta(\delta)$ of a transition function $\delta(s)$, such that:

$$\zeta(\delta; x) = \begin{cases} 0 & , \forall \xi \in \zeta : \xi(x) = 0, \\ 1 & , \forall \xi \in \zeta : \xi(x) = 1, \\ -1 & , \exists \xi_1 : \exists \xi_2 : \xi_1(x) = 1 \text{ and } \xi_2(x) = 0, \xi_1 \neq \xi_2 \end{cases} \tag{11}$$

Using sets of solutions $\zeta(\delta)$ a GBA transition function $\delta'(s)$ is calculated as:

$$\wedge_{i, \zeta(\delta, s_i) = 1} \, \delta(s_i). \tag{12}$$

Constructed transitions functions (12) may contain redundant transitions. For example, for a transition function like $s_1 \vee s_2$ the disjunctive normal form over

ordinary Boolean functions would be $s_1 \wedge \neg s_2 \vee \neg s_1 \wedge s_2 \vee s_1 \wedge s_2$. However, the use of a disjunctive normal form over positive Boolean functions assumes that there is a transition in one of states $s_1$ or $s_2$ only. After removing redundant transitions on steps 19-21 (Fig. 3) the *reverse* function is used to obtain GBA transitions finally:

$$reverse\big(\zeta\big) = \bigwedge_{i=1}^{m} \rho\big(\zeta(\delta; p_i)\big) \wedge \bigwedge_{i=1}^{n} \rho\big(\zeta(\delta; s_i)\big),$$

where

$$\rho\big(\zeta(\delta; x)\big) = \begin{cases} \neg x & , \zeta(\delta; x) = 0, \\ x & , \zeta(\delta; x) = 1, \\ 1 & , \zeta(\delta; x) = -1 \end{cases}$$

buildGBA($\delta(z^0)$) // $\delta(z^0) = \delta(s^0)$ — transition function of initial state of GBA
1 $\delta_0 \leftarrow \delta(z^0)$
2 $\Delta' \leftarrow \{\delta\}$
3 **foreach** $\zeta(\delta_0)$ :
4 $\delta_1 \leftarrow \wedge_{i,\zeta(\delta_0;s_i)=1}\delta(s_i)$
5 $\Delta' \leftarrow \Delta' \cup \{\delta_1\}$
6 $old\_size \leftarrow 0$
7 $new\_size \leftarrow |\Delta'|$
8 **while** $new\_size \neq old\_size$
9 $old\_size \leftarrow new\_size$
10 **foreach** $\delta_0 \in \Delta'$
11 similar steps 3-5
14 $new\_size \leftarrow |\Delta'|$
15 $\Delta \leftarrow \{\}$
16 **foreach** $\delta_0 \in \Delta'$ // remove redundant transitions
17 $\delta_1 \leftarrow false$
18 **foreach** $\zeta(\delta_0)$ :
19 **for** $i \leftarrow 1$ **to** $n$
20 **if** $\zeta(\delta_0; s_i) = -1$ **then**
21 $\zeta(\delta_0; s_i) \leftarrow 0$
22 $\delta_1 \leftarrow \delta_1 \vee reverse\big(\zeta(\delta_0)\big)$
23 $\Delta \leftarrow \Delta \cup \{\delta_1\}$

**Fig. 3.** Algorithm for constructing the generalized Büchi automaton from a given VWAA

*Example 2.* The initial state of the alternating automaton $B_\varphi$ is $s_5$ (Fig. 2), so the initial state of the corresponding GBA is $z_0 = s_5$. According to $\delta(s_5)$ there are a transition to $s_5$ and a transition with universal choice to $s_2$ and $s_5$, so $z_1 = s_5 \wedge s_2$. There are no new states occurring from the conjunction of transitions $s_2$ and $s_5$. The GBA constructed by our symbolic algorithm is shown in Fig. 4.

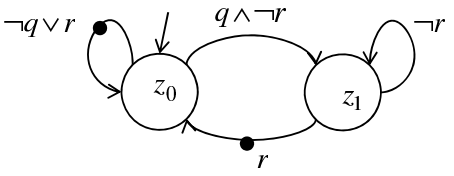



**Fig. 4.** The generalized Büchi automaton with accepting transitions for the formula $\mathrm{G}(q \rightarrow \mathrm{F}r)$

According to the conditions (9-10) accepting transition labeles are defined as:

$$\lambda_k = \neg f_k \vee \exists f_k : (\delta(f_k) \wedge \neg f_k) \tag{13}$$

We use these labels to mark degeneralizer transitions in the algorithm of degeneralizer construction at the next step.

## 7 From Generalized Büchi Automata with Accepting Transitions to Büchi Automata

A Büchi automaton is built as a product of GBA with accepting transitions and an automaton-template, so-called a *degeneralizer* [5]. The number of degeneralizer states depends linearly on the number of GBA accepting labels. The structure of the degeneralizer guarantees that the word is accepting by automata composition if and only if transitions from every GBA accepting group are used. The degeneralizer is used to transfer accepting labels from transitions to states so that there was only one group of accepting labels in the final Büchi automaton. We use the algorithm of degeneralizer construction given in [5] and proved in [3].

The Büchi automaton for the formula $\mathrm{G}(q \rightarrow \mathrm{F}r)$ coincides with $GBA$ where the state $z_0$ is accepting (Fig. 4).

## 8 Results and Related Works

In spite of the fact that in the worst case the number of states of a Büchi automaton is growing exponentially from the LTL formula length, in many cases algorithms based on alternating automaton lead to a very compact Büchi automaton.

For example, the LTL to Büchi algorithm based on atoms (sets of LTL subformulas) and obligations is desrcibed in [7]. For the formula $(p \vee q)\mathrm{U}(p \wedge q)$ the resulting Büchi automaton generated according to the algorithmin [7] has 6 states, while for our algorithm it has only 2 states (Fig. 5).

The idea of using the alternating automata as an intermediate step of building Büchi automata for LTL formula is presented in some another

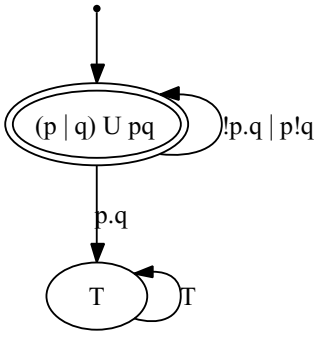



**Fig. 5.** The Büchi automaton constucted by symbolic algorithm for LTL formula $(p \vee q)\mathrm{U}(p \wedge q)$

algorithms [3,4]. The LTL2BA algorithm [3] use explicit presentation of states and transitions of VWAA, GBA and Büchi automata in memory. It uses various rules to minimize the automata by merging the equivalent states and removing the redundant transitions on-the-fly on each step. Also it provides a posteriori Büchi automata simplification. In our symbolic algorithm these simplifications are reached automatically thank to the BDD representation of Boolean characteristic functions. The paper [4] presents a generalized definition of alternating automata with fin- and inf-accepting conditions, which requires new rules for on-the-fly simplification.

Our realization is written in C++ using BuDDy v2.4 library [11] used for operations over BDD. We compare our realization with the Büchi generator of Spin model checker [8], one of the most popular software for LTL verification, developed by Bell Labs, and the LTL2BA program, which realizes an explicit version of the algorithm based on alternating automata described in [3]. Those both programs are written in C too. All tests were done on Intel Core 2 Duo CPU (2.33 GHz) with 2 GB of RAM.

Consider a formula $\mathrm{G}(q \rightarrow \mathrm{F}r)$, which means that a request $q$ always leads a response $r$ in the future. This kind of formulas refining with fairness conditions are often encountered in practice:

$$\Phi_n = \neg\big((\mathrm{GF}p_1 \wedge ... \wedge \mathrm{GF}p_n) \rightarrow \mathrm{G}(q \rightarrow \mathrm{F}r)\big).$$

The experiments results are presented in Table 1. It is practically impossible to use Spin to generate Büchi automata for this kind of formulas with more than four fairness conditions. Moreover, the number of states and transitions of Büchi automata generated by Spin is significantly greater than automata generated with algorithms based on alternating automata. The LTL2BA program reaches unreasonable time of work after nine fairness conditions.

**Table 1.** Time results for the formula $\Phi_n$ for $1 \leq n \leq 11$, time is in *sec*

| $n$ | Spin | LTL2BA | Symbolic |
|---|---|---|---|
| 1 | 0.05 | $< 0.01$ | $< 0.01$ |
| 2 | 0.19 | $< 0.01$ | $< 0.01$ |
| 3 | 4 | $< 0.01$ | $< 0.01$ |
| 4 | 155 | $< 0.01$ | $< 0.01$ |
| 5 | 4607 | 0.05 | 0.03 |
| 6 | 5232 | 0.57 | 0.11 |
| 7 | 8113 | 4 | 0.33 |
| 8 | 11212 | 45 | 2 |
| 9 | + | 375 | 11 |
| 10 | + | 4500 | 16 |
| 11 | + | $> 36000$ | 36 |

The same results has been shown by tests with the following kind of formulas (Table 2):

$$\Psi_n = \neg\big(p_1 \mathrm{U}(p_2 \mathrm{U}(...\mathrm{U}p_n)...)\big).$$

**Table 2.** Time results in *sec* on the formula $\Psi_n$ for $2 \leq n \leq 10$

| $n$ | Spin | LTL2BA | Symbolic |
|---|---|---|---|
| 2 | 0.02 | $< 0.01$ | $< 0.01$ |
| 3 | 0.03 | $< 0.01$ | $< 0.01$ |
| 4 | 0.17 | $< 0.01$ | $< 0.01$ |
| 5 | 1.23 | $< 0.01$ | $< 0.01$ |
| 6 | 38 | 0.02 | $< 0.01$ |
| 7 | 127 | 1.15 | 0.02 |
| 8 | + | 150 | 0.03 |
| 9 | + | $> 3600$ | 0.17 |
| 10 | + | + | 0.63 |

The Büchi generator of the Spin model checker fails out of memory on formula $\Psi_8$.

The paper [3] presents the results of comparison LTL2BA program with algorithms LTL2AUT [9] and Wring [10], which improve Spin's algorithm. Spin and LTL2AUT use the optimization ideas: to generate states by demand only and use state labels rather than transition labels. In addition Wring simplifies formulas before translation using NNF. LTL2BA has shown the best results in described comparison. According to our experiments (Tables 1 and 2) the symbolic algorithm works more than in a thousand times faster than LTL2BA for some $n$.

## 9 Conclusion

In recent years, the model checking as the method for improving the quality of parallel and distributed programs is actively developed to find more effective algorithms and to apply it to real practical software systems. In this paper we consider one of the algorithms included in the LTL model checking.

The theoretical basis for using alternating automata for translation of linear time logic (LTL) formulas in Büchi automata is given in [1,2,3]. The advantage of this approach stems from the fact that in most cases the resulting Büchi automaton is rather compact. However, the translation algorithm itself poses exponential requirements on processing time and memory.

We used binary decision diagrams (BDD) to represent all data structures and all operations performed on them as binary functions. This representation led to significantly reduced complexity of the process of Büchi automaton construction for some types of LTL formulas.

## References

1. Muller, D., Schupp, P.E.: Alternating Automata on Infinite Objects, Determinacy and Rabin's Theorem. In: Perrin, D., Nivat, M. (eds.) Proc. Automata on Infinite Words – École de Printemps d'Informatique Théorique (EPIT 1984). LNCS, vol. 192, pp. 99–107 (1985)
2. Vardi, M.: Nontraditional Applications of Automata Theory. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 575–597. Springer, Heidelberg (1994)
3. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
4. Tauriainen, H.: Automata and Linear Temporal Logic: Translations with Trace-Based Acceptance. PhD.Thesis, Helsinki University of Technology (2006)
5. Giannakopoulou, D., Lerda, F.: From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002)
6. Rohde, G.S.: Alternating Automata and the Temporal Logic of Ordinals. PhD. Thesis in Mathematics, University of Illinois at Urbana-Champaign (1997)
7. Karpov, Y.G.: Model Checking: Verification of Parallel and Distributed Program Systems, p. 560. SPb:BHV-Petersburg (2010) (in Russian)
8. Holzmann, G.: Spin Model Checker. The Primer and Reference Manual, p. 608. Addison-Wesley, Reading (2003)
9. Daniele, M., Giunchiglia, F., Vardi, M.: Improved automata generation for linear temporal logic. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 249–260. Springer, Heidelberg (1999)
10. Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
11. The BuDDy Project, http://sourceforge.net/projects/buddy/

# Sisal 3.2 Language Features Overview*

Alexander Stasenko

A.P. Ershov Institute of Informatics Systems
astasenko@gmail.com

**Abstract.** This paper contains a short introduction of Sisal language and an overview of features introduced by Sisal 3.2 version compared to Sisal 3.1 version. Sisal 3.2 features a multidimensional array support, new abstractions like parametric types and generalized procedures, more flexible user-defined reductions, an improved interoperability with other programming languages and a specification of several optimizing source text annotations. Sisal 3.x version is used as an input language of a system of functional programming (SFP).

**Keywords:** functional programming, dataflow languages, scientific computations.

## 1 Introduction

Imperative programming languages and their traditional extensions like OpenMP [1] are not very convenient for a parallel program development because they require a low level specification of parallelism that can lead to subtle program errors that are hard to detect and fix. In addition, existing popular technologies for a parallel program specification such as OpenMP and nVidia CUDA [2] often rely on a specific machine architecture (e.g. OpenMP is designed to produce SMP[1] friendly code and CUDA was designed for nVidia GPUs) thus these technologies are not suitable for a portable specification of parallelism.

To overcome the above mentioned limitations in imperative languages other extensions such as OpenCL [3] and Intel Parallel Building Block are introduced. For example Intel Parallel Building Blocks augment C++ language with technologies like Threading Building Blocks [4] library for task-centric parallelism, Array Building Blocks for data-centric parallelism and Intel Ct [5] that combines task and data flow parallelism. These technologies allow developer to be more independent of hardware architecture as for example some work can be seamlessly offloaded to GPU.

There are several areas where parallelism is very important. One is game programming (visualization, physics) and other is scientific computations. C/C++ language that is targeted by most parallel extensions is quite suitable for programming games however scientific world is much more conservative and Fortran

* Work is partly supported by the Russian Foundation for Basic Research (RFBR grant no. 07-07-12050).

[1] Acronym SMP stands for Symmetric Multiprocessing.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 110–124, 2011.

programming language [6] is still quite popular there. Since Fortran is also an imperative language that is designed to specify computations for a sequential Von Neumann architecture it is not easy to write parallel programs in Fortran even despite some its inherently parallel features such as built-in arrays.

Sisal programming language [7] was created to be a successor of Fortran language and to be more suitable for a parallel programming [8]. To achieve this Sisal was designed to be functional programming language with types and constructions that allow easy parallelization (that is why it is often called a dataflow language) on variety of machine architectures including quite exotic dataflow supercomputers. To ease transformation of imperative style Fortran programs Sisal contains different kinds of loop expressions that in fact are quite unusual in a world of functional languages. Sisal computations are always deterministic and can be described in a form of acyclic dataflow graph where nodes represent operations and edges represent data. Sisal supports exception handling in a form of special error value that every type contains.

It was demonstrated that Sisal performance need not be worse than programs written in imperative languages [8, 9] as it was for example demonstrated with controlled comparison on real-world image processing benchmark code [10] and other applications such as a Gauss-Jordon linear equation solver, a particle in cell simulation, a protein simulation program [11], the Lawrence Livermore Loops [12], a SIMPLE hydrodynamics code [13] and a one level barotropic weather simulation [14]. The acceptable Sisal performance was reached after some experiments with different forms of algorithms in Sisal language that take into account Sisal implemented compiler optimizations, not with different parallelization techniques. In contrast the imperative program exposes a variety of different parallelization techniques which are independent of algorithm. In addition it was showed that parallel code in imperative language was considerably large then the Sisal code since a large component of parallel imperative code is related to control overheads.

Sisal 3.2 programming language [15] introduced by this paper is a successor of Sisal 3.1 [16] language that was developed in IIS SB RAS. Sisal 3.1 integrated the most important features of Sisal 90 [17] and Sisal 3.2 integrated features of Sisal 2.0 [18] version. This paper contains an overview of features introduced by Sisal 3.2 version compared to Sisal 3.1 version. Sisal 3.x version is used as an input language of a system of functional programming (SFP) [19]. SFP aims to provide programmer with a convenient parallel program development environment on his personal computer and seamless transfer of his program to supercomputer environment without need for its adaptation.

This paper is organized in a following way. The section 2 describes the general features of Sisal language. The section 3 describes multidimensional arrays that came from Sisal 2.0 language. The section 4 describes new language abstractions such as parametric types and generalized functions and operations. The section 5 describes new way to specify user-defined reductions that were introduced in Sisal 90. The section 6 describes the way Sisal 3.2 programs can interoperate with other programming languages. The section 7 describes existing optimizing annotations or pragma statements.

## 2 Sisal Language Overview

This section describes general features of Sisal 3.x language without going to details which can be found in corresponding reference manuals.

### 2.1 Program Structure

An Sisal program consists of one or more separately compilation units called *modules*. Each module consists of *definition* and *declaration* files. Module declaration file corresponds to one module definition and each module definition can not have more than one module declaration.

Sisal module contains definition and declaration of procedures (functions and operations), types and contract definitions. Module declaration contains procedure declarations which are defined by the corresponding module definition and are visible outside it. In addition module declaration contains externally visible declarations and definitions of types and contracts.

Any function in any module may be the starting point of program execution. At this outermost level, function parameters are values obtained from the operating system level, an function results are produced at that level.

Since Sisal compiler translates Sisal programs into C programming language, all Sisal function definitions have corresponding C language equivalent definitions which in turn have corresponding declaration in C language which allows a software not written in Sisal to have subsidiary parts written in Sisal. Special foreign module declarations declares the relationship between Sisal and a set of subsidiary code written in other languages. This allows Sisal software to access libraries of already written code.

### 2.2 Types

Data types include the usual scalar types (boolean, character, integer, real, double), structured types (records and unions, arrays and streams) and functions. Structured types may have values of any type as components; records and unions have heterogeneous components and arrays and streams have homogeneous components. Unions can be recursive like in the following example:

```
type list[T] := union [ empty;
                        item: record [value: T; next: list] ]
```

The language supports user defined types with their custom operations thus for example allowing programmer to implement complex number types. This is an example of a definition of a complex number type and its additive operation:

```
type complex := record [ real_part, imag_part: real ]
operation +(complex, complex returns complex)
```

A module declaration may specify the name of a record or union type for public use, but may prevent exportation of the components.

In the previous versions of Sisal language a type may be declared to be one of a set of alternatives. This is useful for writing functions with formal parameter types not given concretely. A function reference will supply known actual parameter types, which are used to complete the compilation of this instance of the function. This facility for *typesets* was used to simplify production of code that operates on different arithmetic types. However in Sisal 3.x version typesets were replaced by a parametric types described in section 4 because they provide greater flexibility and do not require delayed compilation. Functionality of recursive typesets can be specified via recursive unions.

Function values may be parameters to functions and the results of expression evaluation, so function types may be declared by giving the types of all parameters and results. Therefore Sisal does not use a complete type inference system wherein the types of all values are inferred from their contexts. As a result complete compile-time typing is possible for all Sisal programs. For example here is declaration of two function types which explicitly specify types of function arguments and return values:

```
// function type of two integer values
// that return two integers results
type footype1 = function [integer, integer returns
                          integer, integer]
// function type without arguments
// and one integer result (constant)
type footype2 = function [returns integer]
```

### 2.3 Functions

A function is declared by listing its name, the names and types of its formal parameters, and types of its result values. The content of a function is one or more expressions (a *multi-expression*) whose type correspond to the result types. Values are available to the expressions via formal parameters, not through globally accessed names.

Higher-order function operations are part of Sisal. Functions can be passed to and returned from functions and be the values of expressions.

### 2.4 Expressions

Expressions are, of course, the heart of the language. Syntax is designed to be as familiar to more traditional procedural languages like Pascal as possible.

**Simple Expressions and Name Scoping.** Conventional infix operations combine scalar arithmetic values. Sisal supports some type promotion automatically and provides some predefined type conversion functions.

One can assign the value of any expression to a name and use the name as shorthand for the expression throughout the scope of the definition. This scoping is done with the **let** construct. For example:

```
// multi-expression equals to 3.0 * G * 3.0, 3.0
let X := 3.0; A := X * G in A * X, X end let
// this expression equals to 4
let A := 3 in let A := A + 1 in A end let end let
```

**Arrays.** Sisal has comprehensive facilities for defining and manipulating array values. An array generator allows the definition of a multidimensional object whose parts form a "tiling" of the overall structure. Arbitrary subarray selection is provided beyond the rectangular subsets available in some other notations. Many infix operations operate element-by-element on array operands and a useful set of functions on arrays is defined. A subarray update facility allows safe alteration of array values. Many applications are expressible succinctly with these features. Array generation, selection and update may use vector subscripts to refer to arbitrary, non geometric sections of arrays.

**Streams.** A stream is a sequence of values produced in order by one expression evaluation and consumed in the same order by one or more other expression evaluations. Producers and consumers are usually **for** expressions but short forms for simple streams are also available. To expose the pipelined parallelism that streams make possible, they must be implemented non-strictly. That is consumer expressions must be started whether or not the producer expression has finished.

### 2.5 Control by Selection

Two constructs for selection are provided in Sisal: **if** and **case** expressions. The results of **if** expression are guarded by boolean expressions, while the **case** expression is guarded by the values of the selecting expression. The arms of a single **if** or **case** expression must agree in arity and type unless the selection is being used for type inquiry.

This example of **if** expression computes roots of a square equation:

```
let d := b**2 - 4*a*c
in if     d > 0 then (-b+d**0.5)/(2*a), (-b-d**0.5)/(2*a)
   elseif d = 0 then          -b/(2*a),          -b/(2*a)
   else                  error[real],       error[real]
   end if
end let
```

This is example of some **case** expression:

```
case die_1 + die_2
   of 2..3,     12 then          "lose"
   of    7,     11 then           "win"
   of 4..6, 8..10 then "no decision"
   else error[array of character]
end case
```

### 2.6 Control by Distribution and Iteration

A single **for** construct has two forms for potentially parallel, as well as sequential, evaluation. In the first form values are distributed to the bodies of the construct and each body defines values to contribute to the overall result. The second form has dependencies (determined by **old** keyword before loop value name) between values defined in one body and used in the successor body. In either form the values from the bodies are collectable into an array or a stream or reducible to a single value. Array construction in **for** expressions allows permutation of the individual body values.

For example this **for** expression computes $\pi$ value iteratively:

```
for Approx := 1.0; Sign := 1.0; Denom := 1.0; i := 1
while i <= Cycles do
  Sign   := - old Sign;
  Denom  := old Denom + 2.0;
  Approx := old Approx + Sign / Denom;
  i      := old i + 1
returns value of Approx * 4.0
end for
```

This **for** expression also computes $\pi$ value but can do it in parallel (since there are no **old** keywords used):

```
for i in 1..Cycles/2 do
  val := 1.0 / (4*i-3):real- 1.0 / (4*i-1):real
returns sum of val
end for * 4.0
```

### 2.7 Errors

Sisal includes standard error processing semantics for managing erroneous computations. However, a Sisal implementation may elect to stop execution when an error is encountered. Each Sisal type has a distinguished value, **error**. Any failed expression evaluation results in **error** of the appropriate type. Error values propagate in a well-defined way when they are operands in computations. Error values can be tested for and even explicitly assigned to signify other anomalous conditions.

## 3 Multidimensional Arrays

Sisal 3.1 has multidimensional arrays in a form of nested one-dimensional arrays of free form (with not specified bounds) which complicates effective array implementation via contiguous array memory layout with direct access. Sisal language was designed to describe scientific computations so after analysis of features of other languages with scientific orientation such as Fortran and Sisal 2.0 it was decided to introduce multidimensional arrays and arrays with fixed form to Sisal

3.2 language as well as extended means for their construction. Multidimensional arrays allow more effective optimizations compared to nested arrays because their rectangular form can be taken into account for array element out-of-bound access checks, which are essential for Sisal language determinism guarantee, as well as for data dependence analysis and transformation [20]. In additional rectangular form of multidimensional arrays is very well suited for representation of such common objects as matrices and images.

Array type is described as "**array** *array form of array element type*". Array can have free "[ *list of double dots* ]" or fixed "[ *list of duplets* ]" form. Duplet is a construction that looks like "*lower bound .. upper bound*", where *lower* and *upper bounds* are unary expressions of an integer type. The lower bound can be omitted and it is assumed to be equal to one by default. The upper bound must be more or equal to the lower bound. Array form can be omitted and is assumed to be " [..] " by default. An array dimension is determined by a dimension of array form which equals to a number of double dots or duplets in it.

Here are some examples of array type declarations:

```
// one-dim array of integers
type Arr1 = array of integer
// two-dim array of integers
type Arr2 = array [.., ..] of integer
// 2 × 3 array of integers
type Arr3 = array [..2, ..3] of integer
```

Array value constructors[2] were also extended to handle arrays of fixed form:

```
// two-dim array [ [1, 0, 3], [4, 0, 6] ]
array [1..2, 1..3] of [1,1 := 1; 1,3 := 3;
                       2,1 := 4; 2,3 := 6; else := 0]
```

To ease construction of large arrays, Sisal 3.2 supports range-based element specification:

```
// range of elements can be specified by a list of values
// like in this array [1, 2, 3]
array [1..3] of [1..3 := 1, 2, 3],
// array range can be specified by integer arrays of indices
// like in this array [0, 2, 1]
array [1..3] of [[3, 2] := 1, 2; else := 0],
// values can be also taken from other array
// like in this array [[1, 2, 3], [4, 5, 6]]
array [..2, ..3] of [1, .. := [1, 2, 3]; 2, .. := [4, 5, 6]]
```

[2] Since Sisal is a functional language, all its values including arrays are constructed at once and, for example, any array element update conceptually leads to creation of a new array value. Array elements not specified by a constructor equal to error values and their ambiguous definition leads to erroneous array.

Sisal 3.2 supports specification of ranges with dimensions tied by **dot** operator[3]:

```
// array [[1, 0, 0], [0, 2, 0], [0, 0, 3]]
array [..3, ..3] of [.. dot .. := 1, 2, 3; else := 0]
```

Dimension range indices can be named and reused later even in the same range (making it triangular):

```
// array [[2, 0, 0], [0, 4, 0], [0, 0, 6]]
array [..3, ..3] of [i in .. dot j in .. := i+j; else := 0]
```

## 4 New Language Abstractions

To increase level of its algorithmic abstractions, Sisal 3.2 was augmented by new conceptions of parametric types, contracts and generalized procedures (functions and operations). A parametric type defines a set of types that allows finer control compared to already existing typesets[4].

A contract is another form of abstraction that allows binding a set of operations over types listed as contract parameters to a contract name. Contracts are used in generalized procedures to specify what kind of operations their parametric types are expected to have. Parametric types, contracts and generalized procedures together specify abstract types with partly defined structure and set of operations. Abstract types such as matrixes and vectors are quite common in mathematics and new functionality allowed to write Sisal 3.2 support libraries for them. Comparing new functionality with other programming languages one can note that parametric type is similair to C++ language class templates without methods, contracts are similair to abstract classes (classes with virtual methods only) and generalized procedures are similair to functions that use classes with virtual methods.

For example let's define a parametric type of matrix over some element type:

```
type matrix[T] := array [.., ..] of T
```

For a matrix multiplication operation, that we are going to declare, a matrix element type must support addition and multiplication operations, so we define a contract with name "additive" for all such types:

```
contract additive[T]
  operation + (T, T returns T)
  operation * (T, T returns T)
end contract
```

[3] Operator **dot** takes two dimension indices $A_{1..m}$ and $B_{1..n}$ that normally produce a Cartesian product of indices and produces a tied sequence of indices $(A_1, B_1)$, $(A_2, B_2)$, ..., $(A_s, B_s)$, where $s = min(n, m)$.

[4] For example a classic typeset cannot define set of records with two fields of equal type while this is possible with parametric type. In addition type names in parametric type can be used in generalized procedures and their contract attachments.

And now we can declare a generalized operation of our matrix multiplication:

```
operation * of additive[T] (matrix[T], matrix[T]
                            returns matrix[T])
```

Generalized procedures require additional runtime support in a form of additional hidden parameter that contains array of pointers to the contract operations (similar to the C++ virtual function tables) that can became known only that generalized procedure call site. As with C++ language this additional overhead was considered an acceptable tradeoff for the increased language flexibility.

## 5 Reusable User-Defined Reductions

In Sisal 3.1 a user-defined reduction was a function definition of a very special form that is used to transform loop values into loop results. Because of its special syntax form a reduction function cannot be reused outside loops. In Sisal 3.2 user-defined reductions are defined as a combination of several usual functions thus allowing them to be reused as normal functions.

A general form of reduction invocation in a loop return statement looks as follows: "**reduction** *name N* ( *list of initial values* ) **of** ( *list of loop values* )" where initial values of reduction must be loop constants. Reduction name $N$ corresponds to functions with prototypes described below.

The first function (function $I$) computes an initial reduction state in a type $T$ which is any type that can hold a reduction internal state:

**function** N ( *types of initial reduction parameters* **returns** T )

The following function (function $L$) recomputes the reduction state using loop values of the subsequent loop iteration:

**function** N ( T, *types of loop reduction parameters* **returns** T )

The following optionally present function (function $J$) determines how some two reduction states (obtained after parallel loop execution) can be merged. This function can be omitted if the reduction does not allow such things. Sisal 3.1 reductions were not able to represent this function $J$, which is essential for parallel reduction computation, so it can be said that Sisal 3.2 introduces parallel reductions if corresponding optimizing annotations are in place[5].

**function** N ( T, T **returns** T )

The last function (function $R$) computes reduction results from its internal state:

**function** N ( T **returns** *types of return reduction values* )

For example these declarations can be used to define sum reduction that adds all loop values and returns zero in case of zero-trip loop[6]:

[5] For more details about optimizing annotations please refer to section 7.

[6] A zero-trip loop is a loop that ends without performing any single iteration.

```
type sum_red[T] := T
contract add1[T]
  operation + (T, T returns T)
  function zero (returns T)
end contract
//$ identity
function sum of add1[T] (returns sum_red[T])
function sum of add1[T] (sum_red[T], T returns sum_red[T])
/*$ commutative */ /*$ associative */
function sum of add1[T] (sum_red[T], sum_red[T]
                         returns sum_red[T])
function sum of add1[T] (sum_red[T] returns T)
function zero (returns integer)
function zero (returns real)
```

Despite the fact that normally a Sisal 3.2 program became large because monolithic Sisal 3.1 reduction is now represented by four more simple functions, this is compensated by increased flexibility as for example some of these functions be foreign function coded in other programming language. Also language became simpler because reductions now do not require special handling.

## 6 Improved Interoperability with Other Languages

Sisal 3.2 language extends Sisal 2.0 language by adding support of foreign language functions from already written programs and libraries (written in C/C++ or Fortran). A foreign functions support is based on a concept of foreign types.

Foreign types in Sisal 3.2 language are specified by their string representation on their native programming language. Values of foreign types are constructed via foreign operations and functions that are written on a foreign programming language and use a special interface to access values of Sisal 3.2 types if necessary. For example with help of foreign types one can define machine specific integer and define operations of implicit type cast between it and built-in Sisal integer type:

```
type int32 = "__int32"
operation (integer returns int32)
operation (int32 returns integer)
```

Foreign types do not have any built-in operations so Sisal program must define them to make use of them. If a foreign type $T$ has an operation "**operation** (T **returns** T)" defined then this operation is used to create a foreign type value copy. For a foreign type $T$ one can prohibit a copy operation at all by using "no **operation** (T **returns** T)" declaration. If a copy operation is not defined and is not prohibited then a bit-by-bit copy approach is used. If a foreign type $T$ has an operation "**operation** (T **returns** null)" defined then it is used to free copy of the foreign type $T$ otherwise no additional actions are performed then a foreign type $T$ value is no longer needed. An error value of a foreign

type $T$ corresponds to it undefined value unless "**operation** (null **returns** T)" operation, that returns an error value of the foreign type, is defined.

This example demonstrates a declaration of a foreign type that corresponds to a matrix type of variable dimensions in C language:

```
// pe points to a dynamic memory
type c_matrix = "struct { int w, h; int* pe; }"
operation (array [..,..] of integer returns c_matrix)
operation (c_matrix returns array [..,..] of integer)
operation (c_matrix returns c_matrix) // copying is possible
// here we free pe memory
operation (c_matrix returns null)
// error[c_matrix] is matrix with zero w and h
operation (null returns c_matrix)
```

A declaration of a foreign function looks like "**function** *function name* ( *list of formal parameters* **returns** *type of return value* )" where at least one formal parameter type or return value type belongs to a typeset $S$. The typeset $S$ contains foreign types, user-defined types based on types from $S$, arrays with elements of type from the typeset $S$, records and non-recursive unions based on types from the typeset $S$. Formal parameters of foreign function can be prefixed by **in**, **out** and **raw** keywords and type of return value can be prefixed by **out** and **raw** keywords.

The keyword **in** can prefix any type $T$ from the typeset $S$. The keyword **in** means that a foreign function receives a pointer to a dynamic memory that contains a copy of value of the type $T$. The keyword **out** can be used together with keyword **in** for the types from a typeset $S_2$. The typeset $S_2$ contains foreign types, arrays with elements of type from the typeset $S_2$, records based on types from the typeset $S_2$. The keyword **out** means that upon return from a foreign function call a new return value is formed from the dynamic memory pointed by pointer passed to the foreign function. If a foreign function has at least one keyword **out** (with or without keyword **in**) then **returns** statement (and a type of return value) can be omitted.

The keyword **out** without the keyword **in** can prefix type $T$ from a typeset $S_3$. The typeset $S_3$ contains foreign types, fixed form arrays with elements of type from the typeset $S_3$ and records based on types from the typeset $S_3$. The keyword **out** before a type $T$ means that a foreign function receives a pointer to a dynamic, not initialized memory of size enough to hold a value of the type $T$ and upon return from the foreign function call a new return value is formed from the dynamic memory pointed by the pointer passed to the foreign function. For the keyword **out** without the keyword **in** the corresponding argument of a foreign function should be omitted.

The keyword **raw** can prefix a type $T$ from a typeset $S_4$. The typeset $S_4$ contains foreign types, records and non-recursive unions based on types from the typeset $S_4$. The keyword **raw** for a foreign type can be used only together with the keyword **in** and it means that this foreign type should be considered as a pointer and it should be passed to the foreign function without any additional

indirection[7]. The keyword **raw** for a record or union means that it will be passed to a foreign function without any extra information that comes with normal Sisal types (such as an error value information). For example the following declarations correspond to C language functions that operate on *c_matrix* type.

```
// corresponds to the following C declaration:
// typedef struct { int w, h; int* pe; } M;
// M mul3(M, M);
function mul3 (c_matrix, c_matrix returns c_matrix)
// corresponds to the following C declaration,
// where resulting matrix is recomputed
// in the first argument: void mul2(M, M);
function mul2 (raw in out c_matrix, c_matrix)
// corresponds to the following C declaration:
// void mul2p(M*, M);
function mul2p (in out c_matrix, c_matrix)
```

Although improved interoperation with other languages does not directly contribute to Sisal performance it for example enables usage math libraries that are optimized for target hardware which can greatly contribute to Sisal program effectiveness. It worth to note that Sisal computation results that depend on foreign function calls can no longer has deterministic property.

## 7 Optimizing Annotations

Sisal 3.2 language is the first version of Sisal language that specifies optimizing annotations in a form of pragma statements. A pragma statement is a special form of comment that starts with dollar sign "$". This section describes all currently recognized pragma statements. Other pragmas are yet to be introduced to improve Sisal 3.2 compiler optimizations that are not yet implemented.

Every expression can be prefixed by a pragma "assert = Boolean expression", that can be controlled for truth after the expression evaluation during program debugging and can be then used for program optimizing transformations. A result of the expression can be denoted as underscore symbol "_" and if the expression is n-ary (where n<1), then its components can be denoted as an array with name "_": "_[1]", ..., "_[n]". In addition, the pragma "assert = Boolean expression" can be placed before **returns** keyword in procedure declarations and can be used to control results of this procedure after its invocation. As an example of the assert pragma statement usage please consider this factorial function declaration and definition:

[7] The keyword **raw** actually makes sense only when keywords **in** and **out** are used together because without keyword **out** keywords **raw** and **in** do not have any effect and can be omitted.

```
// factorial of number n
forward function fact (n: integer
/*$ assert = n >= 1*/ /*$ assert = _ >= n*/
  returns integer)
function fact (n: integer returns integer)
  if n = 1 then 1
           else /*$ assert = _ > 0*/ fact(n-1)*n end if
end function
```

Another pragma "parallel" can be used before a case expression in Sisal (analogous to a switch expression in C language). This pragma can be specified if it is known that only one test can be true or in case of pragma "parallel = Boolean expression" only one test is true if the specified Boolean expression is true.

Functions that are used to form a reduction value (see section 5) can be associated with special pragmas that can be used to parallel loops more efficiently. A function $I$ that computes an initial reduction state can be marked by pragma "identity" if it specifies an identity value, relative to functions $L$ and $J$, that merges reduction states of a type $T$: $J(t, L(I(I_1), L_1)) = L(t, L_1)$[8], where $t$ is in $T$, $I_1$ is initial values and $L_1$ is loop values. If the function $I$ is not marked by the identity pragma then it does not make much sense to define the function $J$ because it would not be able to correctly merge reduction states.

The function $J$ marked by "associative" pragma is assumed to be associative: $J(J(a,b),c) = J(a,J(b,c))$. The function $J$ marked by "commutative" pragma is assumed to be commutative: $J(a,b) = J(b,a)$. If the function $J$ is associative then it may be used for parallel computation of reduction values[9]. Therefore it does not make much sense to define non-associative function $J$. If the function $J$ is associative and commutative then it can be used for a potentially more effective asynchronous parallel computation of reduction values. Asynchronous parallel computation is then function $J$ is used for values $L_n$ and $L_m$ computed in any order (where $|n-m| \geq 1$): $J(L(I(I_1), L_n), L(I(I_1), L_m))$. The example which covers usage of "identity", "associative" and "commutative" pragmas can be found in the section 5.

A declaration of a foreign procedure can be prefixed by a pragma "weight = integer expression", where a value of integer expression defines an approximate number of cycles of some abstract machine required to execute this procedure. A weight ratio is used for better load balancing of foreign procedures between several processing units.

[8] This equality comes from the requirement to represent sequential computation of two loop values $L(L(I(I_1), L_1), L_2)$ as composition of two computations that can be executed in parallel: $J(L(I(I_1), L_1), L(I(I_1), L_2))$.

[9] If function $I$ specifies identity value and function $J$ is associative then we can divide sequential computations of any length $L(\ldots L(L(I(I_1), L_1), L_2), \ldots, L_3), L_4)$ into $J \ldots (J(L(I(I_1), L_1), L(I(I_1), L_2)), \ldots, J(L(I(I_1), L_3), L(I(I_1), L_4)) \ldots)$ computation that can be computed in parallel in a tree-like manner.

## 8 Conclusion

Sisal 3.2 language is a significant improvement over previous Sisal 3.1 version. Sisal 3.2 became closer to modern programming languages after introduction of multidimensional arrays, parallel reductions, new type and algorithmic abstractions, improved interoperability with other programming languages and specification of optimizing annotations. Multidimensional arrays with fixed form allow more effective implementation via contiguous array memory layout with direct access. New type and algorithmic abstractions are introduced to specify abstract types with partlydefined structure and set of operations which are quite common in mathematics. Improved interoperability with other programming languages allow to better and at fuller extent reuse already written libraries of scientific source code. As a result Sisal 3.2 became more convenient for a scientific programming and Sisal optimizing compiler.

## References

[1] Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press, Cambridge (2007)
[2] Sanders, J.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, New York (2010)
[3] Tsuchiyama, R., Nakamura, T., Iizuka, T., Asahara, A., Miki, S.: The OpenCL Programming Book, Kindle edn. Fixstars Corporation (2010)
[4] Reinders, J.: Intel Threading Building Blocks. O'Reilly Media, Cambridge (2007)
[5] Ghuloum, A., Smith, T., Wu, G., Zhou, X., Fang, J., Guo, P., So, B., Rajagopalan, M., Chen, Y., Chen, B.: Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture. Intel Technology Journal 11(4), 333–348 (2007)
[6] ISO/IEC 1539-1:2004(E).: Information technology: Programming languages: Fortran: Part 1: Base language. Internat. Organization for Standardization (ISO), Central Secretariat, Geneva (2004)
[7] McGraw, J.R., Skedzielewski, S.K., Allan, S.J., Oldehoeft, R.R., Glauert, J., Kirkham, C., Noyce, B, Thomas, R.: Sisal: Streams and iterations in a single assignment language, Language Reference Manual, Version 1.2. Technical report, Lawrence Livermore National Laboratory, Livermore (1985)
[8] Cann, D.C.: Retire Fortran?: a debate rekindled. Communications of the ACM 35(8), 81–89 (1992)
[9] Cann, D.: Vectorization of an Applicative Language: Current Results and Future Directions. Technical report, Lawrence Livermore National Laboratory, Livermore (1990)
[10] Abramson, D., Mckay, A.: Evaluating the Performance of a SISAL implementation of the Abingdon Cross Image Processing Benchmark. International Journal of Parallel Programming 23(2), 105–134 (1995)
[11] Cann, J.R., York, E.J., Stewart J.M., Vera, J.C., Maccioni, R.B.: Small zone gel chromotography of interacting systems: Theoretical and experimental evaluation of elution profiles for kinetically controlled macromolecule-ligand reactions. Analytical Biochemistry (1988)

[12] McMahon, F.H.: The Livermore Fortran kernels: A computer test of the numerical performance range. Technical report, Lawrence Livermore National Laboratory, Livermore (1986)
[13] Crowley, W.P., Henderson, C.P., Rudy, T.E.: The simple code. Technical report, Lawrence Livermore National Laboratory, Livermore (1978)
[14] Chang, P., Egan, G.: An Implementation of a Barotropic Numerical Weather Prediction Model in the Functional Language SISAL. In: Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (1990)
[15] Kasyanov, V.N., Stasenko, A.P.: Sisal 3.2 programming language. In: Methods and instruments for program construction, pp. 56–134. IIS SB RAS, Novosibirsk (2007) (in Russian)
[16] Stasenko, A.P., Sinyakov, A.I.: Basic features of Sisal 3.1 programming language. IIS SB RAS, Novosibirsk (2006) (in Russian)
[17] Feo, J.T., Miller, P.J., Skedzielewski, S.K, Denton, S.M.: Sisal 90 users guide. Lawrence Livermore National Laboratory, Livermore (1995)
[18] Cann, D.C., Feo, J.T., Bohm, A.P.W, Oldehoeft, R.R.: Sisal Reference Manual: Language Version 2.0. Technical report, Lawrence Livermore National Laboratory, Livermore (1991)
[19] Kasyanov, V.N., Stasenko, A.P., Gluhankov, M.P., Dortman, P.A., Pyjov, K.A., Sinyakov, A.I.: SFP - An interactive visual environment for supporting of functional programming and supercomputing. WSEAS Transactions on Computers 5(9), 2063–2070 (2006)
[20] Allen, R., Kennedy, K.: Automatic translation of FORTRAN programs to vector form. ACM Transactions on Programming Languages and Systems (TOPLAS) 9(4), 491–542 (1987)

# A Cellular Automata Based Model for Pedestrian and Group Dynamics: Motivations and First Experiments

Stefania Bandini[1,2], Federico Rubagotti[1], Giuseppe Vizzari[1,2], and Kenichiro Shimura[3]

[1] Complex Systems and Artificial Intelligence (CSAI) research center
Department of Computer Science, Systems and Communication (DISCo)
Università degli Studi di Milano - Bicocca, Viale Sarca 336/14, 20126 Milano, Italy
{bandini,vizzari}@csai.disco.unimib.it,
f.rubagotti@campus.unimib.it
[2] Crystals Project, Centre of Research Excellence in Hajj and Omrah (Hajjcore)
Umm Al-Qura University, Makkah, Saudi Arabia
[3] Research Center for Advanced Science & Technology, The University of Tokyo, Japan
shimura@tokai.t.u-tokyo.ac.jp

**Abstract.** The simulation of pedestrian dynamics is a consolidated area of application for cellular automata based models: successful case studies can be found in the literature and off-the-shelf simulators are commonly employed by end-users, decision makers and consultancy companies. These models represent pedestrians as agents, but the overall system dynamics is determined simplistically: agents uniformly tend to reach the destination without colliding with obstacles and other pedestrians. Aspects like (i) the impact of cultural heterogeneity among individuals and (ii) the effects of the presence of groups and particular relationships among pedestrians are generally neglected or underestimated. This work describes a cellular automata based model, introducing an innovative behavioral model that encapsulates the theory of proxemics and a simplified representation of the influences determined by the presence of groups of pedestrians in the crowd. A simple scenario is reproduced to observe the influences on the pedestrian dynamics determined by the presence of groups in the crowd and to evaluate the implications of some modeling choices. Results are discussed and compared to experimental observations and to data available in the literature.

## 1 Introduction

There are several features of crowds of pedestrians suggesting that they can be considered as complex entities: the mix of competition for a shared space and the collaboration due to the (not necessarily explicit but generally implied) social norms, the dependency of individual choices on the past actions of other individuals and on the current perceived state of the system, the possibility to detect self-organization and emergent phenomena, they are all indicators of the intrinsic complexity of a crowd. The activities of architects, designers and urban planners are deeply influenced by the study of the movement of pedestrians in built environments (see, e.g., [2] and [13]), which is, in turn, determined by numerous elements both human and environmental. It is also necessary to study how different situations, both normal and extraordinary, influence the

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 125–139, 2011.

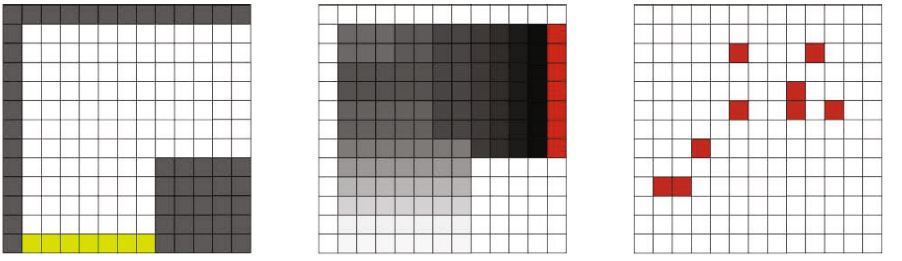
human behavior, especially considering dramatic episodes such as terrorist attacks, riots and fires, but also due to the growing issues in facing the organization and management of public events (ceremonies, races, carnivals, concerts, parties/social gatherings, and so on) and in designing naturally crowded places (e.g. stations, arenas, airports). Computational models for the simulation of crowds are thus growingly investigated in the scientific context, and these efforts led to the realization of commercial off-the-shelf simulators often adopted by firms and decision makers[1]. Even if research on this topic is still quite lively and far from a complete understanding of the complex phenomena related to crowds of pedestrians in the environment, models and simulators have shown their usefulness in supporting architectural designers and urban planners in their decisions by creating the possibility to envision the behaviour/movement of crowds of pedestrians in specific environments, to elaborate what-if scenarios and evaluate their decisions with reference to specific metrics and criteria.

Cellular Automata [14] have been widely adopted as a conceptual and computational instrument for the simulation of complex systems (see, e.g., [12]); in this specific context several CA based models (see, e.g., [10,3]) have been adopted as an alternative to particle-based approaches [6], and they also influenced new approaches based on autonomous situated agents (see, e.g., [5,7,1]). The main aim of this work is to present GA-Ped (Group Aware pedestrians), a CA based model for pedestrian and crowd dynamics for a multidisciplinary investigation of the complex dynamics that characterize aggregations of pedestrians and crowds. This work is set in the context of the Crystals project[2], a joint research effort between the Complex Systems and Artificial Intelligence research center of the University of Milano–Bicocca, the Centre of Research Excellence in Hajj and Omrah and the Research Center for Advanced Science and Technology of the University of Tokyo. The main focus of the project is on the adoption of CA and agent based approaches to pedestrian and crowd modeling to investigate meaningful relationships between the contributions of anthropology, cultural characteristics and existing results on the research on crowd dynamics, and how the presence of heterogeneous groups influence emergent dynamics in the context of the Hajj and Omrah. The last point is in fact an open topic in the context of pedestrian modeling and simulation approaches: the implications of particular relationships among pedestrians in a crowd are generally not considered or treated in a very simplistic way by current approaches. In the specific context of the Hajj, the yearly pilgrimage to Mecca that involves over 2 millions of people coming from over 150 countries, the presence of groups (possibly characterized by an internal structure) and the cultural differences among pedestrians represent two fundamental features of the reference scenario. Studying implications of these basic features is the main aim of the Crystals project.

The paper is organized as follows: the following we will introduce the CA based pedestrian and crowd model considering the possibility of pedestrians to be organized in groups, while Sect. 3 summarizes the results of the application of this model in a simple simulation scenario. Conclusions and future developments will end the paper.

[1] See, e.g., Legion Ltd. (`http://www.legion.com`), Crowd Dynamics Ltd. (`http://www.crowddynamics.com/`), Savannah Simulations AG (`http://www.savannah-simulations.ch`).

[2] `http://www.csai.disco.unimib.it/CSAI/CRYSTALS/`



(a) First Layer: geometry and obstacles in dark gray, generator for pedestrians coming from the south exit in green.

(b) Second Layer: floor fields (one for each point of interest). In this example the floor field representing the path to the eastern exit.

(c) Third Layer: the position of pedestrians inside the environment

**Fig. 1.** The separation of the environment into three layers in a L-shaped corridor configuration

## 2 Definition of the GA-Ped Model

We now introduce some principles considered during the definition of the GA-Ped model. We decided to simulate the interactions between pedestrians in an environment that is discrete both in space and in time. We introduced a two-dimensional cellular automata (CA) structure with local interactions, and a discrete-time dynamical system to model the movements of pedestrians inside a structured environment. We chose a discrete approach in order to achieve an efficient implementation for fast computer simulation, while maintaining a sufficient expressiveness in the definition of the rules for pedestrian movement. Moreover, the model employs floor fields (see, e.g., [4]) to support pedestrian navigation in the environment: each relevant final or intermediate target contained in the scenario is associated to a floor field, a sort of gradient indicating the most direct way towards the associated point of interest.

Our system is composed of the triple: $Sys = \langle Env, Ped, Rules \rangle$. The first element to be introduced is the environment: it contains the representation of different objects (e.g. walls, obstacles, etc.) and, during the simulation, pedestrians. Pedestrians can observe their neighborhood, looking for the best path to reach the targets specified in a schedule. Every pedestrian is endowed of an internal state, that is a memory used to save the schedule, feelings, past actions and their characterization. The last element of our model is a set of transition rules, determining the evolution of the system.

Now we introduce our model in detail, starting from the representation of space and environment. Then we focus the attention on the modeling of pedestrians, concluding with details on the update rules necessary to determine the dynamics of the system.

### 2.1 Space and Environment

The representation of the space in our model is based on the Cellular Automata theory. It is split into squared cells with fixed width, obtaining a two-dimensional grid. Namely, in our model the space is discretized into small cells which may be empty or occupied by exactly one pedestrian. At each discrete time step it is possible to analise the state of the system by observing the state of each cell (and, consequently, the

position of each pedestrian into the environment). In our model the environment is defined as $Env = \langle Space, Fields, Generators \rangle$, where the $Space$ is a physical, bounded bi-dimensional area where pedestrians and objects are located; the size of the space is defined as a pair of values $(xsize, ysize)$ and it is specified by the user. In our model we consider only rectangular-shaped scenarios (but it is possible to define different shapes suitably employing non walkable cells). The space in our model is modeled using a three-layer structure: $Space = \langle l_1, l_2, l_3 \rangle$ where each layer represents details related to a particular aspect of the environment. As represented in Fig. 1, each layer is a rectangular matrix sharing the same size of the other two. The first layer $(l_1)$, contains all the details about the geometry of the environment and the properties of each cell. A cell may be a generator (i.e. a cell that can generate new pedestrians according to some associated parameters), and can be walkable or not. A cell is thus characterized by a $cellID$, an unique key for each cell, and, if the cell can generate pedestrians it can be associated to a *generating spot* (a set of generator cells located in the same area). The *second layer*, denoted as $l_2$, contains information about the values of the floor fields of each cell. Each cell contains an array of pairs $(floorID, value)$, one for each target. The *third layer*, $l_3$, is made up of cells that may be empty or occupied by one pedestrian. This layer stores the position of each pedestrian. The aim of this partitioning is to branch three different domains of information into three different views in order to keep our model cleaner, easier to understand and implement.

**Generators and Targets.** Information about generators and targets are saved into the first and second layer. Generators are cells that, at any iteration, may generate new pedestrians according to predetermined rules. *Generating spots* are groups of generator cells located in the same area and driven by the same set of rules of generation. In our model a *generating spot* is defined as follows:

$$spot = \langle spotID, maxPed, positions, groups, itineraries, frequency \rangle$$

where $spotID$ is an identifier for the generator; $maxPed$ is the maximum amount of pedestrians that the spot can generate during the entire simulation; $positions$ indicate the cells belonging to that generating spot; $groups$ being the set of group types that can be generated, each associated with a frequency of generation; $itineraries$ that can be assigned to each pedestrian, considering the fact that group members share the same schedule but that different groups may have different schedules, each associated with a frequency; $frequency$ is a value between 0 and 100, specifying the frequency of pedestrian generation (0 means never generate pedestrians, 100 means generate pedestrians at each iteration, if free space is available and if the desired maximum population has not been reached).

A target is a location in the environment that the pedestrians may desire to reach, due to its position or to the presence of a particular object. Examples of targets in a train station are ticket machines, platforms, exits, lounges and so on. A traveller may have a complex schedule composed of different targets like: (a) I have to buy a ticket, then (b) I want to drink a coffee and (c) reach platform number 10 to board the train to Berlin. This plan can be translated in the following schedule made of points of interest located inside the environment: (i) ticket machine, (ii) lounge, (iii) platform 10. From now on the words *schedule* and *itinerary* are used interchangeably as (in our framework) they

define the same concept. We will describe how pedestrians will be able to move towards the target later on.

**Floor Fields.** As stated previously, the floor field can be thought of as values saved in a grid underlying the primary grid of the environment. Each target has a floor field and the corresponding values are saved into the $l_2$ of the environment. A floor field contains information suggesting the shotest path to reach the associated destination. Floor field values are distributed in every cell of the environment. In our model each cell contains information about every target defined in the model. Given the cell at position $(x, y)$, the corresponding floor field values are saved in $l_2(c_{x,y})$ as a list of pairs with the following structure: $(floorID, value)$. Values of a floor field are integers between 0 and 256. Given a target, if a cell has a floor field value 0 for that particular destination, means that no indications to reach the target is available. On the contrary, if the value of the cell is 256 means that the target has been accomplished (because the target is in that cell).

We can distinguish between two classes of floor fields: *static* and *dynamic*. The *static floor field* does not evolve with time and it is not influenced by the presence of pedestrians. The *dynamic floor field* is modified by the presence of pedestrians and it is updated using two procedures called *diffusion* and *decay*. In our model we have only *static* floor fields, specifying the shortest path to destinations and targets. Interactions between pedestrians that in other models are described by the use of *dynamic floor fields*, in our model are modeled through a perception model based on the idea of *observation fan*, which will be introduced in Sect. 2.3. An example of floor field is presented in Fig. 1.b. A greyscale is used to visually show its values: darker cells have higher floor field values, the position of the target is highlighted in red. It is possible to observe that cells near the target have higher values. Floor field values influence the transition probabilities of a pedestrian, as a person usually will try to follow the shortest path to the target.

### 2.2 Time and Update Type

Our model is a discrete-time dynamical system, and update rules are applied to all pedestrians following an update method called *shuffled sequential update* [8]. At each iteration, pedestrians are chose following a random sequence and then updated. This choice was made in order to implement our method of collision avoidance based on cell reservation. In the shuffled sequential update, a pedestrian, when choosing the destination cell, has to check if this cell has been reserved by another pedestrian within the same time step. If not, the pedestrian will reserve that cell, ant it will move into at the end of the iteration. If the cell is already reserved, an alternative destination has to be chosen.

Each iteration corresponds to an amount of time directly proportional to the size of the cells of the environment and to the reaction time: given a squared cell of $40 \times 40cm^2$, the corresponding timescale is approximately of $0.3sec$ of real time, obtained by transposing the empirically observed value of average velocity of a pedestrian, that is $1.3m/s$ to the maximal walking speed of one cell per time step [11].

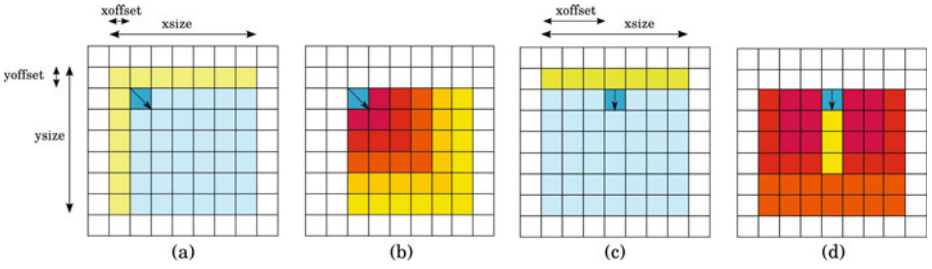



**Fig. 2.** Example of the shape of an observation fan for a diagonal direction (in this case south-east) and for a straight direction (in this case south): (a and c) in light cyan the cells that are observable by the pedestrian and are used for the evaluation, in green the observable backward area; (b and d) the weight matrix applied for the evaluation, in this case objects or pedestrians near the pedestrian have more weight than farther ones (e.g. this fan is useful for evaluating walls).

### 2.3 Pedestrians

We now focus the attention on the modeling of pedestrians: first we introduce how they are represented. Pedestrians are modeled as the state of cells in a bidimensional grid. Each pedestrian is provided with some attributes describing details such as group membership, ID, schedules. Then we introduce the perception model: each pedestrian is endowed with a set of *observation fans* that determines how they see and evaluate the environment. Attributes, internal state and environment influence the behavior of our pedestrians: movement decisions are modeled using a Finite State Automata and a set of rules. In detail, a pedestrian can move in one of the cells belonging to its Moore neighborhood, and to any possible movement is associated a revenue value, called *likability*, representing the desirability of moving into that position. While in the previous Sect. we introduced the notion and structure of simulation turn, we will now show how a single pedestrian action is performed. In the following we will introduce how pedestrians decide their movements, but we already clarify that the two main tasks they perform to choose their destinations: they observe the environment and internal state to obtain the *spatial awareness*; then they evaluate the *likability* of the possible movements to choose the solution that maximizes the benefits.

**Pedestrian Characterization.** We decided to reduce the characterization of our pedestrians to a small set of essential attributes and, in particular:

$$pedestrian = \langle pedID, groupID, schedule \rangle$$

with $pedID$ being an identifier for each pedestrian, $groupID$ (possibly null, in case of individuals) the group the pedestrian belongs to and $schedule$ a list of goals to be accomplished by the pedestrian (one of the above introduced itineraries).

**Perception model.** In our model every pedestrian has the capability to observe the environment around him, looking for other pedestrians, walls and objects. Perception capabilities are modeled with the idea of *observation fan*. An *observation fan* can be considered as the formalization of physical and conceptual perceptive capabilities: it

determines how far a pedestrian can see and how much importance has to be given to the presence of obstacles and other pedestrians. An *observation fan* is similar to the idea of *neighborhood* of a cell in the CA theory as it defines the shape of the observable area, and how to evaluate the observed *things* according to their distance from the pedestrian. An *observation fan* is defined as follows:

$$\zeta = \langle type, xsize, ysize, weight, xoffset, yoffset \rangle$$

where:

- $type$ identifies the direction of the fan: it can be 1 for diagonal directions and 2 for straight directions (the fan has different shapes and it may be asymmetric);
- sizes and offsets are defined as shown in Fig. 2. Sizes ($xsize$ and $ysize$) define the maximum distance to which the pedestrian can see. The shape of the fan is influenced by both the direction and the sizes. The offsets are used to define if the pedestrian can see backward and the size of the lateral view (only type 2, see Fig 2.c);
- $weight$ is a matrix of values $w_{x,y} \in \mathbb{R}_+$ defined in the interval $[0, 1]$. These values determine the relationship between the *thing* that has been observed and the distance (e.g. the distance of a wall influences differently the movement of a pedestrian).

For each class of pedestrians is possible to define multiple *observation fans*; each fan can be applied when evaluating walls, pedestrians belonging to the same group, to other groups or, lastly, to particular groups. For instance, this feature is useful when modeling situations like football matches: it is possible to define two classes of groups, one made of supporters of the first team and the other of supporters of the second team. Groups belonging to the first class will interact differently if dealing with other groups belonging to the first class or bel onging to the second one.

**Behavior and Transition Rules.** In this Sect. we introduce the evaluation phases and the transition rules that model the pedestrian behavior. First, we introduce our concept of pedestrians modeled as *Deterministic Finite Automata*[3] (DFA). Then we focus the attention on the behavior of the pedestrians in our model. It is determined by different aspects, like the minimization of the time necessary to reach the destination, the need to maintain a significative distance from strangers while preserving the cohesion of the group and the necessity to avoid obstacles. The decision of a movement is taken after an evaluation of the environment and the internal state, choosing the best tradeoff of the aspects previously introduced.

*Pedestrian states and transitions.* The behavior of a pedestrian is represented as a flow made of four stages: *sleep*, *context evaluation*, *movement evaluation*, *movement*.

When a new iteration is started, each pedestrian is in a sleeping state. This state is the only possible in this stage, and the pedestrian does nothing but waits for a trigger signal from the system. The system wakes up each pedestrian once per iteration and,

[3] We are not modeling all the features of a Deterministic Finite Automaton: we are not recognizing languages and we do not have accepting states.

then, the pedestrian passes to a new state of context evaluation. In this stage, the pedestrian tries to collect all the information necessary to obtain spatial awareness. When the pedestrian has collected enough data about the environment around them, it reaches a new state. In this state behavioral rules are applied using the previously gathered data and a movement decision is taken. When the new position is notified to the system, the pedestrian returns to the initial state and waits for the new iteration.

In our model, pedestrian active behavior is limited to only two phases: in the second stage pedestrians collect all the information necessary to recognize the features of the environment around him and recall some data from their internal state about last actions and desired targets. A first set of rules determine the new state of the pedestrian. The new state, belonging to the stage of movement evaluation, depicts the current circumstances the pedestrian is experiencing: e.g. the situation may be normal, the pedestrian may be stuck in a jam, it may be compressed in a dense crowd or lost in an unknown environment (i.e. no valid floor field values associated to the desired destination). This state of awareness is necessary to the choice of the movement as different circumstances may lead to different choices: a pedestrian stuck in a jam may try to go in the opposite direction in search for an alternative path, a lost pedestrian may start a random walk or look for other significant floor fields.

We represent pedestrian behavior with a DFA; in particular, our automaton $M$ is a 4-tuple $(Q, E, \delta, q_0)$, where:

- $Q$ a list of states;
- $E$ a list of events;
- $\delta : Q \times E \rightarrow Q$ a transition function;
- $q_0 \in Q$ an initial state.

The set of states $Q$ is partitioned into four subsets:

1. $Sleeping$: only one state (sleeping);
2. $ContextEvaluation$: only one state, the pedestrian is collecting data to achieve *spatial awareness*;
3. $MovementEvaluation$: the pedestrian is aware of its situation and it is evaluating all the possible alternatives;
4. $Movement$: nine states belong to this subset, one for each direction;

Also the events belonging to $E$ are partitioned into four subsets, as every event can be associated to only one pair of states.

**Pedestrian Movements.** We now focus the attention on the modeling of how our pedestrians evaluate the possible movements and how they choose the best movement.

*Direction and speed of movement.* At each time step, pedestrians can change their position along nine directions (keeping the current position is considered a valid option), into the cells belonging to their Moore neighborhood of range $r = 1$. Each possible movement has a value called *likability* that determines how much the move is *good* in the terms of the criteria previously introduced.

In order to keep our model simple and reduce complexity, we do not model different movement speeds. At each iteration a pedestrian can move only in the cells belonging

to the Moore neighborhood, reaching a speed value of 1 or it can maintain the position (in this case speed is 0)[4].

*Functions and notation.* In order to fully comprehend the pedestrian behavior introduced in the following paragraphs, it is necessary to premise the notational conventions and the functions we have introduced in our modelization:

- $c_{x,y}$ defines the cell with (valid) coordinates $(x, y)$;
- $Floors$ is the set of the targets instantiated during the simulation. Each target has a floor field and they share the same *floorID* (i.e. with $t \in Floors$ we define both the target and the associated floor field);
- $Groups$ the set containing the *groupID* of the groups instantiated during the simulation dynamics;
- $Classes$ is the set containing all the group classes declared when defining the scenario;
- $Directions$ is the set of the possible directions. Are nine, defined using cardinal directions: $\{N, NE, E, SE, S, SW, W, NW, C\}$.

Given $x \in [0, xsize - 1]$ and $y \in [0, ysize - 1]$, we define some functions useful to determine the characteristics and the status of the cell $c_{x,y}$:

- **cell walkability**: this function determines if the cell $c_{x,y}$ is walkable or not (e.g. if there is a wall). If the cell is walkable the function returns the value 1, otherwise it returns 0. It is defined as follows:

$$l_1(c_{x,y}) = [0, xsize - 1] \times [0, ysize - 1] \rightarrow \{0, 1\} : \\ 0 \text{ if the cell is not walkable,1 otherwise;} \quad (1)$$

- **floor field value**: this function determines the value of the floor field $t$ in the cell $c_{x,y}$. If the cell contains the target associated to the target $t$, the function returns the value 256. If there is no floor field available for the target $t$ the function returns the value 0. If a valid floor field is present the function return its value, which is defined in the interval $[1, 255]$:

$$l_2(c_{x,y}, t) = [0, xsize - 1] \times [0, ysize - 1] \times t \in Floors \rightarrow [0, 256] : \\ 0 \text{ if the floor field for } t \text{ is not available, } 256 \text{ if} \\ \text{the cell is the target, the floor field value for } t \text{ otherwise;} \quad (2)$$

- **presence of pedestrians belonging to a particular group** this function determines if in the cell $c_{x,y}$ contains a pedestrian belonging to a particular group $g$ specified as input. If a pedestrian belonging to that group is contained in the cell, the function returns 1, otherwise it returns 0:

$$l_3(c_{x,y}, g) = [0, xsize - 1] \times [0, ysize - 1] \times g \in groups \rightarrow [0, 1] : \\ 0 \text{ if the cell does not contain a pedestrian} \\ \text{belonging to the group } g, 1 \text{ otherwise.} \quad (3)$$

[4] Our pedestrians can move only to the cells with distance 1 according to the Tchebychev distance.

*Observation fan.* We define $\zeta_{x,y,d}$ as the set of cells that are observable according to the characteristics of the observation fan $\zeta$, used by a pedestrian located in the cell at coordinates $(x, y)$ and looking in the direction $d$.

The overall *likability* of a possible solution can be thought as the desirability of one of the neighboring cells. The more a cell is desirable, the higher is the probability that a pedestrian will choose to move into that position. In our model the *likability* is determined by the evaluation of the environment and it is defined as a composition of the following sequence of characteristics:

$$\begin{array}{lccccccc} likability = & goal\ driven + & group & + proxemic + & geometrical + & stochastic \\ & component & cohesion & repulsion & repulsion & contribution \end{array}$$

Formally, given a pedestrian belonging to the group class $g \in Groups$, in the state $q \in Q$ and reaching a goal $t \in Floors$, the *likability* of a neighbouring cell $c_{x,y}$ is defined as $li(c_{x,y})$ and is obtained evaluating the maximum benefit the pedestrian can achieve moving into this cell (following the direction $d \in Directions$) using the observation fan $\zeta$. The value of the characteristics that influence the likability are defined as follow:

- **goal driven component:** it is the pedestrian wish to quickly reach its destination and it is represented by the floor field. Our model follows the least effort theory: pedestrians will move on the shortest path to the target which needs the least effort. This component is defined as $l_2(c_{x,y}, t)$: it is the value of the floor field in the cell at coordinates $(x, y)$ for the target $t$;
- **group cohesion:** we want to preserve group cohesion, minimizing the distances between the members of the group. This component is defined as the pedestrians belonging to the same group in the observation fan $\zeta$, evaluated according to the associated weight matrix:

$$\zeta(group, d, (x, y), g) = \sum^{c_{i,j} \in \zeta_{x,y,d}} w^{\zeta}_{i,j} \cdot l_3(c_{i,j}, g) \tag{4}$$

- **geometrical repulsion:** it represents the presence of walls and obstacles. Usually a pedestrian wishes to avoid the contact with these objects and the movement is consequently influenced by their position. This influence is defined as the presence of walls (located in layer $l_1$) inside the observation fan $\zeta$, according to the weight matrix for $walls$ specified in the same observation fan:

$$\zeta(walls, d, (x, y)) = \sum^{c_{i,j} \in \zeta_{x,y,d}} w^{\zeta}_{i,j} \cdot l_1(c_{i,j}) \tag{5}$$

- **proxemic repulsion:** it is the repulsion determined by the presence of strangers, both individuals or belonging to other groups. A pedestrian whishes to maintain a *safe* distance from these pedestrians and this desire is defined as the sum of these people in the observation fan $\zeta$, according to the weight matrix for the group of these pedestrians:

$$\zeta(strangers, d, (x, y), g) = \sum^{c_{i,j} \in \zeta_{x,y,d}} w^{\zeta}_{i,j} \cdot (1 - l_3(c_{i,j}, g)); \tag{6}$$

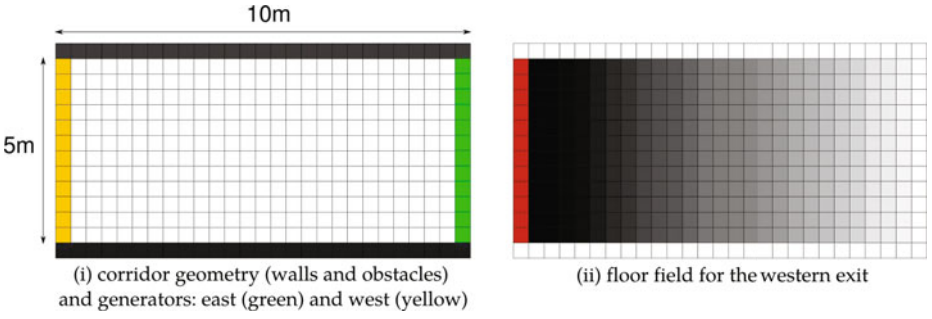



**Fig. 3.** Representation of the corridor scenario: environment geometry, generators and floor fields

– **stochasticity:** similarly to some traffic simulation models (e.g. [9]), in order to introduce more realism and to obtain a non deterministic model, we define $\epsilon \in [0, 1]$ as a a random value that is different for each *likability* values and introduces stochasticity in the decision of the next movement.

Formally, these four environmental influences, plus the fifth element of stochasticity, compose the $likability$ of a movement as follows:

$$li(c_{x,y}, d, g, t) = j_w \zeta(walls, d, (x, y)) + j_f field(t, (x, y)) - j_g \zeta(group, d, (x, y), g) - j_n \zeta(strangers, d, (x, y), g) + \epsilon. \quad (7)$$

Group cohesion and floor fields are positive components because they positively influence a decision as a pedestrian wishes to reach the destination quickly, keeping the group cohese at the same time. On the contrary, the presence of obstacles and other pedestrians has a negative impact as pedestrians usually tend to avoid this contingency.

The formula 7 summarizes the evaluation of the aspects that characterize the *likability* of a solution. A pedestrian *opens* an observation fan for each possible movement, and it examines the environment in the corresponding directions, evaluating the elements that may make that movement opportune (e.g. the presence of other pedestrians belonging to the same group or an high floor field value and data that may discourage as the presence of walls or pedestrians belonging to other groups).

## 3 Simulation Scenario

The simulated scenario consists in a rectangular corridor, $5m$ wide and $10m$ long. We assume that the boundaries are open and that walls are present in the north and south borders. The width of the cells is $40cm$ and the sizes of the corridor are represented with 14 cells and 25 cells respectively. Pedestrians are generated at the east and west borders and their goal is to reach the opposite exit. Floor fields, environment geometry and generators are graphically represented in Fig. 3.

We investigated the capability of our model to fit the fundamental diagram proposed in the literature. As shown in Figure 4 our model correctly represents the nature of pedestrian dynamics: if the frequency of generation is low, consequently the flow is

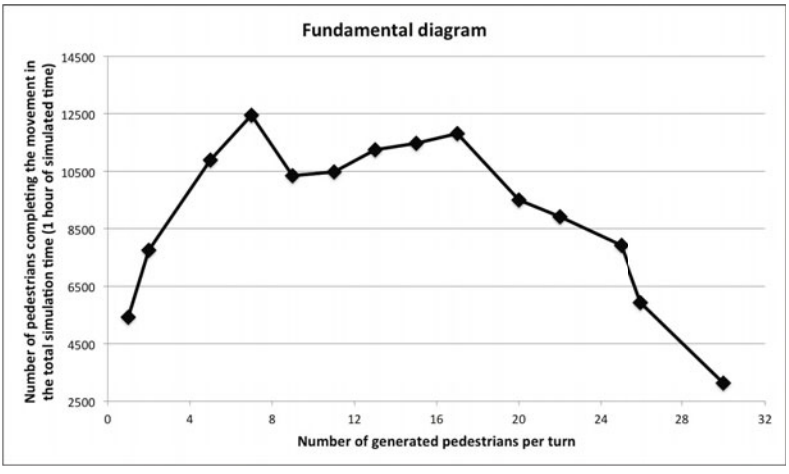



**Fig. 4.** Fundamental diagram for the rectangular corridor with groups of size 3. The density is specified as frequency of generation. The ratio between members of groups and alone pedestrians is 40/60.

low. Increasing the frequency leads to a higher throughput until a critical density has been reached. If the system density is increased beyond that value, the flow begins to decrease significantly as the friction between pedestrians makes movements more difficult. Observing the same figure we can state that, before the critical density has been reached, the flow is fluid, similar to the laminar flow that can be see in the models of traffic simulation. After the value of critical density has been reached, the simulations underline a greater variability in the fundamental diagram. In fact, at higher density the possibility of events that may disrupt the flow are more frequent causing a sensible variation of the throughput.

We also performed additional qualitative observations of the dynamics generated by the model and we can state that it is capable to generate the following phenomena:

- lane formation at high densities;
- the higher is the number and the size of the groups into the environment, the lower will be the total flow, due to the higher degree of friction between different groups.

### 3.1 Large Group vs. Small Group Counterflow

We were interest in studying the dynamics of friction and avoidance that are verified when two groups with different size, traveling in opposite directions, are facing each others in a rectangular shaped corridor. We simulated the $5m \times 10m$ corridor with one large group traveling from the left (west) to the right (east), opposed to one small group traveling in the opposite direction. The aim of this particular set up was to investigate the differences in the dispersion of the smaller group with respect of the size of the large group and the overall time necessary to walk through the corridor. From now on we call the small group as the *challenging* group and the large group as the *opponent* group.

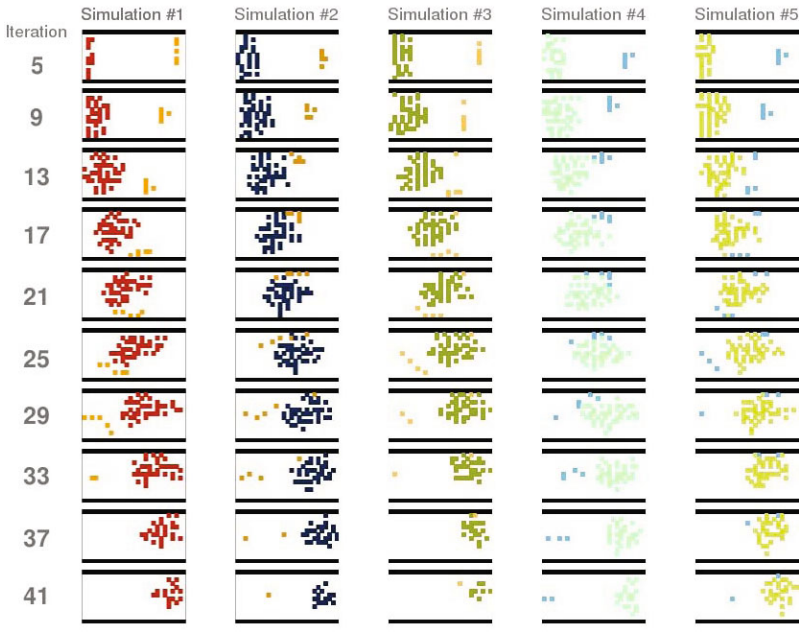



**Fig. 5.** Images representing the state of the simulation taken at different time steps. The opponent group is composed of 50 pedestrians, while the challenging group size is 5.

We considered opponent group of five different sizes: 10, 20, 30, 40 and 50. Challenging groups were defined with only two sizes: 3 and 5. The results are consistent with the observable phenomena as the model can simulate all the three possible cases that can be spotted in the real world:

- the challenging group remains compact and moves around the opponent group;
- one or more members of the challenging group moves around the larger group in the other side with respect to the other members of the group;
- one or more members of the challenging group remain stuck in the middle of the opponent group and then the small group temporarily breaks up.

It is also interesting to point out that in our model, if a split is verified in the challenging group, when their members overcome the opponent group, they aim to form again a compact configuration. The actual size of the simulation scenario is however too small to detect this *reforming* of the group[5]. Figure 5 shows some images representing the state of the simulation at different time steps. As stated before, it is possible to observe the range of different circumstance that our model is able to simulate: for example in the simulation #1 the challenging groups can overcome the opponent one simply by moving around it, the same situation is represented in simulation #2 and #4 but the challenging group experiences more friction generated by the opponents. In the same

[5] We carried out additional simulations in larger environments and we qualitatively observed the group re-union.

figure, the simulation #3 and #5 show a challenging group that splits in two and their members moving around the opponent group on both the two sides.

Finally, we investigated the relationships between the time necessary to the members of the challenging group to reach the opposite end of the corridor in relation with the size of the opponent group. As expected, and in tune with the previous observations, the larger the size of the opponent group, the higher time necessary to the members of the challenging group to reach their destination is. The difference of size in the challenging group only slightly influences the performances: it is easier to remain stuck in the opponent group but the difference between three and five pedestrians is insufficient to obtain significant differences.

## 4 Conclusions and Future Developments

The paper presented a CA based pedestrian model considering groups as a first-class element influencing the overall system dynamics. An original model considering a simple notion of group (i.e. a set of pedestrians sharing the destination of their movement and the tendency to stay close to each other) has been presented and applied to a simple scenario, gathering results that are in tune with the existing literature on this topic. Validation against real data is being conducted and preliminary results show a promising correspondence between simulated and observed data. Future works are aimed at a concrete application of the model in the context of the Crystals project and further extensions of the notion of group and related dynamics.

**Acknowledgments.** This work is a result of the Crystals Project, funded by the Centre of Research Excellence in Hajj and Omrah (Hajjcore), Umm Al-Qura University, Makkah, Saudi Arabia.

## References

1. Bandini, S., Federici, M.L., Vizzari, G.: Situated cellular agents approach to crowd modeling and simulation. Cybernetics and Systems 38(7), 729–753 (2007)
2. Batty, M.: Agent based pedestrian modeling (editorial). Environment and Planning B: Planning and Design 28, 321–326 (2001)
3. Blue, V.J., Adler, J.L.: Cellular automata microsimulation for modeling bi-directional pedestrian walkways. Transportation Research Part B: Methodological 35(3), 293–312 (2001)
4. Burstedde, C., Klauck, K., Schadschneider, A., Zittartz, J.: Simulation of pedestrian dynamics using a two-dimensional cellular automaton. Physica A: Statistical Mechanics and its Applications 295(3-4), 507–525 (2001)
5. Dijkstra, J., Jessurun, J., Timmermans, H.J.P.: A Multi-Agent Cellular Automata Model of Pedestrian Movement. In: Pedestrian and Evacuation Dynamics, pp. 173–181. Springer, Heidelberg (2001)
6. Helbing, D., Molnár, P.: Social force model for pedestrian dynamics. Phys. Rev. E51(5), 4282–4286 (1995)
7. Henein, C.M., White, T.: Agent-based modelling of forces in crowds. In: Davidsson, P., Logan, B., Takadama, K. (eds.) MABS 2004. LNCS (LNAI), vol. 3415, pp. 173–184. Springer, Heidelberg (2005)

8. Klüpfel, H.L.: A Cellular Automaton Model for Crowd Movement and Egress Simulation. Ph.D. thesis, Universität Duisburg-Essen (July 2003)
9. Rickert, M., Nagel, K., Schreckenberg, M., Latour, A.: Two lane traffic simulations using cellular automata. Physica A: Statistical and Theoretical Physics 231(4), 534–550 (1996)
10. Schadschneider, A., Kirchner, A., Nishinari, K.: Ca approach to collective phenomena in pedestrian dynamics. In: Bandini, S., Chopard, B., Tomassini, M. (eds.) ACRI 2002. LNCS, vol. 2493, pp. 239–248. Springer, Heidelberg (2002)
11. Weidmann, U.: Transporttechnik der Fussgänger. Schriftenreihe des IVT 90 ETH Zürich (1992)
12. Weimar, J.R.: Simulation with Cellular Automata. Logos Verlag, Berlin (1998); ISBN 3-89722-026-1
13. Willis, A., Gjersoe, N., Havard, C., Kerridge, J., Kukla, R.: Human movement behaviour in urban spaces: Implications for the design and modelling of effective pedestrian environments. Environment and Planning B 31(6), 805–828 (2004)
14. Wolfram, S.: Computation theory of cellular automata. Communications in Mathematical Physics 96, 15–57 (1984)

# Using Multi Core Computers for Implementing Cellular Automata Systems*

Olga Bandman

Supercomputer Software Department
ICM&MG, Siberian Branch, Russian Academy of Sciences
Pr. Lavrentieva, 6, Novosibirsk, 630090, Russia
bandman@ssd.sscc.ru

**Abstract.** A concept of cellular automata system (CA-system) is introduced as a model of comp[lex phenomena in which several interacting species are involved. CA system suggests a common work of several CA where each processes its own cellular array using in its transition rules cell states of others CA of the system. Taking into account that multi core computers with shared memory are nowadays widely used, a temptation to accelerate the computation by allocating each CA of the system onto one of computer cores is quite natural. Hence, it would be helpful to know what speedup can be obtained by such a parallelization. The paper is aimed to get an answer to this question by determining the conditions, when multicore parallel implementation of CA systems is expedient and correct, and develop the parallelization algorithms for typical CA systems. The results are illustrated by simulation experiments.

## 1 Introduction

Cellular Automata (CA) being regarded as a model of spatial dynamics, gradually changes its status of object of study for the status of the method for studying natural processes. CA properties such as nonlinearity of transition functions and irreversibility of evolution made them particularly useful for investigating the behavior of complex systems [1], exhibiting self organization and emergency. The number of such investigations increases rapidly, comprising the study of new more complicated phenomena in biology, physics and chemistry [2]. Many of such phenomena are simulated by parallel composition of several CA [3], the evolution of each CA component simulating the corresponding species behavior. Parallel CA composition is a set of CA, operating in common in such a way that each CA transition functions variables are cell states of any CA of the system. By analogy to partial differential equations of traditional numerical analysis, parallel composition of CA are further called *CA systems.* Nowadays CA systems are used mostly in scientific investigation being implemented on personal computers, many computational experiments on one and the same CA being performed

* Supported by (1) Presidium of Russian Academy of Sciences, Basic Research Program N 2 (2009), (2) Siberian Branch of Russian Academy of Sciences, SBRAS Interdisciplinary Project 32 (2009), (3) Project RFBR 11-01-00567a.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 140–151, 2011.

with many different parameters, which should be easily and promptly changed. Although the computational time is wanted to be as small as possible, there is no need to run the programs on remote powerful clusters, each time waiting for the results. But, having a two-, four- or eight-core computer with a shared memory on the table, it is reasonable to make the cores operate in parallel, in correspondence with the parallel composition of CA in the system. Whether it is worth to be done and what are the conditions for such a parallelization be efficient, is the subject of the paper.

The paper is organized as follows. Next section presents the method of parallel composition which provides correctness conditions conservation. In the third section parallel algorithms for two-core implementation of a single reaction-diffusion process simulation is given. Fourth section is devoted to parallel multi-core implementation of a CA system, simulating several interacting reaction-diffusion processes. In the Conclusion the results are summarized and application perspectives are outlined.

## 2 Formal Definition of a Cellular Automata System

In general case CA-system suggests any number $n$ of CA working in common. But since formal definitions for arbitrary $n$ are very cumbersome and hardly comprehensive, for clearness and without loss of generality, the system $\aleph = \Upsilon(\aleph_1, \aleph_2)$ with $n = 2$ is further considered. Each component $\aleph_k$, $k = 1, 2$, is determined by three sets, $\aleph_k = \langle A_k, X_k, \Theta_k \rangle$, where $A_k$ is a state alphabet, $X_k$ - a set of cell names (coordinates in finite discrete space), and $\Theta_k$ – a local operator. The alphabets $A_1, A_2$ may be different and of any type (Boolean, real, symbolic). Both synchronous and asynchronous modes of operation are allowed.

Between $X_1$ and $X_2$, $i = 1, 2 \ldots, I$, $I = |X|$, there exists an one-to one correspondence $\xi : X_1 \to X_2$:

$$\begin{array}{ll} x_2 = \xi(x_1), & \forall x_2 \in X_2, \\ x_1 = \xi^{-1}(x_2), & \forall x_1 \in X_1. \end{array} \tag{1}$$

The elementary entity of a CA is a *cell* represented by a pair $(v, x)$, where $v \in A$, $x \in X$, the state of the cell $x$ being denoted as $v(x)$ or $v_x$. The set $\Omega = \{(v, x)\}$ containing $|X|$ cells with different names forms a *cellular array.* A pair of cells in $\Omega_1 \cup \Omega_2$, such that $x_1 = \xi^{-1}(x_2)$ are further referred to as *twin-cells.* When a cell of any component CA is meant, it is named simply as $x$ . Similarly, in all expressions valid for all CA components, the bottom indices are removed.

In $X_1$ and $X_2$ two types of templates $T(x)$ are defined as follows. Let $d(x_i, x_j)$ be a distance between $x_i$ and $x_j$, $x_i, x_j \in X_k$. Then

$$T_{kk}(x_i) = \{x_j : d(x_i, x_j) < r,\ x_j \in X_k\} \tag{2}$$

represents a *base template* with radius $r$, and

$$T''_{kl}(x_i) = \{x_j : d(\xi(x_i), x_j) < r_l,\ x_i \in X_k,\ x_j \in X_l\},\ k \neq l,\ r_l << |X|, \tag{3}$$

is a *remote context* template with radius $r_l$.

The set of states

$$V(T(x)) = \{v(x_j) : x_j \in T(x)\} \tag{4}$$

form a *local configuration*, with underlying template $T(x)$. The set of states of all cells in $\Omega_k = \{(v, x_i) : \forall x_i \in X_k\}$ is referred to as *global configuration* $V(X_k) = \{v(x_i) : \forall x_i \in X_k\}$.

Each $\Theta_k$ is expressed by a substitution [4] as follows.

$$\Theta_k(x): \quad V(T_{kk}(x)) \star V^{(}T''_k(x)) \rightarrow V'(T_{kk}(x)), \quad k, l = 1, 2. \tag{5}$$

The template $T''_k(x)$ contains two parts, i.e,

$$T''_k(x) = T''_{kk}(x) \cup T''_{kl}(x), \tag{6}$$

where $T''_{kk}(x)) \subset X_k$ is a *self context* and $T''_{kl}(x)) \subset X_l$ is a *remote context*.

The local configuration $V(T_{kk}(x))$ in (5) is called a *base* of $\Theta_k(x)$. Its states are to be replaced by $V'(T_{kk}(x))$, when $\Theta_k(x)$ is applied. The local configuration $V(T''_k(x))$ is called a *context*. Its states are not changed by $\Theta_k(x)$ application, but serve as variables in the *transition functions* $f_{kj}$, whose values $v'(x_j)$ are states in the *next state local configuration* $V'(T_{kk}(x))$:

$$v'(x_j) = f_{kj}(V(T_{kk}(x) \cup T''_k(x)), \quad x_j \in T_{kk}. \tag{7}$$

An application of $\Theta_k$ to a cell $x \in X_k$ means substituting of states $v'(x_j) \in V'(T_{kk}(x))$ for $v(x_j) \in V(T_{kk}(x))$.

If a CA system is a parallel composition $\aleph = \Upsilon(\aleph_1, \aleph_2)$, the two cellular arrays are processed in parallel by application $\Theta_1$ to $\Omega_1$, and $\Theta_2$ to $\Omega_2$. The whole process of simulation consists of a sequence of *iterations*. An iteration presumes that in both CA the corresponding operator has been applied to all cells, which yields a *global transition* of the system: $\Omega_1(t) \cup \Omega_2(t) \rightarrow \Omega_1(t+1) \cup \Omega_2(t+1)$.

In a CA system any mode of operation is allowed: synchronous CA may interact with an asynchronous one, provided correctness conditions for both and for the system in a whole are satisfied, assuming each CA operates according to the following algorithms.

Synchronous CA performs a global transition from $\Omega(t)$ to $\Omega(t+1)$ as follows:

1) $\Omega(t)$ is copied to $\Omega'(t)$.

2) $\Theta(x)$ is applied to all cells from $\Omega'(t)$, next values $v'(x)$ being computed according to (7) and written in twin-cells of $\Omega(t)$.

Asynchronous CA performs a global transition by executing $|X|$ times the following steps:

1) A cell is randomly chosen from $X$.

2) The next state $v'_i(x)$ is computed by (7), and the substitution (5) is immediately performed.

## 3 Correctness of CA System Functioning

CA simulation is considered as an effective computation [5], if it possesses the following properties *safeness* and *fairness*. The first provides any datum not being lost during the simultaneous application of $\Theta_k(x_i)$ and $\Theta_k(x_j)$. A sufficient condition for that is formally expressed as follows:

$$\big(T_{kk}(x_i) \cup T''(x_i)\big) \cap T_{kk}(x_j) = \emptyset \quad \forall x_i, x_j \in X_k. \tag{8}$$

The second property guarantees that all cells of $X_k$ have equal rights to be chosen for $\Theta_k$ application. Synchronous CA, functioning according to the algorithm (sec.2), satisfy safeness condition, if it has a single-cell base, i.e. if $|T_{kk}| = 1$. If it is not so, the CA should be transformed into a superposition of $|T_{kk}|$ single-cell base CA according to a method given in [4]. At any case, the next states of $\Theta_k(x_i)$ application to $\Omega_k$ should be written to an additional array $\Omega'_k$ which is intended for storing the remote contexts. This prevents to change cell states in the left-hand side of (8) that have not been used by application of $\Theta_k(x_j)$, $x_i \neq x_j$. Fairness is satisfied since all cells in $X$ are chosen for being processed with equal probability.

As distinct to synchronous case, in asynchronous CA transition functions (7) are allowed to be applied both to current and to next states and, hence, multi-cell base in local operators are frequently used. So, to guarantee safeness the only requirement is that local operator $\Theta_k(x)$ is to be indivisible, i.e. nothing is allowed to occur between changing the states of $V'(x)$ during its execution. This fact makes useless the additional array when CA operates alone. Fairness is guaranteed by using equal random distribution when choosing cells for $\Theta$ application.

Besides the requirement of all system CA correctness, there are some additional conditions to be met for the whole system functioning be safe and fair.

1) Application of $\Theta_k$ to any cell of $\Omega_l$, $k \neq l$, should be forbidden, otherwise a state might be lost for being used by another application.

2) During the application of $\Theta_k(x)$ to $\Omega_k$,when states from $V(T_{kk}(x))$ are sequentially changing, no cell of $T_{kk}(x)$ shouldbe used as remote context for any $\Theta_l$. Otherwise some states of $\Theta''_l$ might be lost.

Formally the above two statements are expressed as follows.

$$\big(T_{kk}(x_i) \cup T''(x_i)\big) \cap T_{ll}(x_j) = \emptyset \quad \forall x_i, x_j \in X_k \cup X_l, \quad k, l = 1, 2. \tag{9}$$

In a more comprehensive form, the correctness conditions of CA-system may be formulated as follows:

1) All CA of the system should be correct, i.e. satisfy (8).

2) If states of $\Omega_k$ serve as a remote context to $\Theta_l$, then there should be a copy of $\Omega_k(t)$, referred to as $\Omega'_k(t)$, whose cell states are included in remote context local configurations $V(T''_{lk})$, in order to satisfy (9).

If in a CA system there is such a CA, say $\aleph_k$, whose cell states are not used in a remote context of any other CA in the system, then $\aleph_k$ operates according to

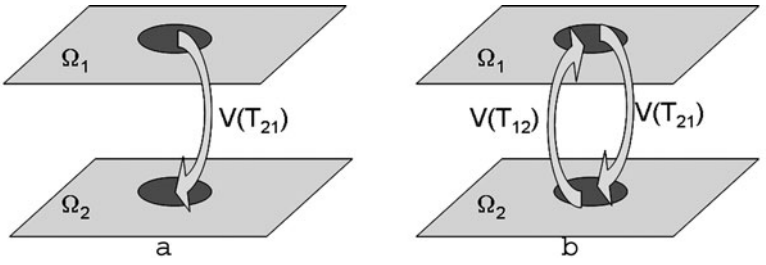



**Fig. 1.** Schematic image of a two types of CA systems: a) when one CA operates independently, b) when both CA are mutually dependent

its mode as if it were alone. In Fig.1 two variants of CA interaction in the system are schematically shown. Moreover, in the case when $\aleph_1$ operates independently and $\aleph_2$ is asynchronous, $\Omega_2'$ is redundant.

## 4 CA-System Simulating a Single Reaction-Diffusion Process

A wide class of phenomena exhibiting complex behavior are dynamical systems where several species move and interact chemically, physically or biologically. Capability of moving is usually independent on reactive interactions, the latter being associated with dissipative (mostly, chemical) character of processes under simulation. Hence, the behavior of each species is described by a pair of typical CA, which are referred to as *diffusion CA* and *reaction CA*. Such a pair in its turn forms a simple diffusion-reaction CA-system, which usually constitutes a *building block* for construction of *complex CA-systems*, simulating several interacting reaction-diffusion processes.

Let $\aleph_1$ and $\aleph_2$ form a reaction-diffusion block simulating diffusion and reaction, respectively (Fig.1a). Parallel algorithm for allocating the system to a multicore processor is as follows.

- Create initial cellular arrays $\Omega_1(0)$ and $\Omega_2(0)$.
- For each iteration $t = 1, \ldots, T$:
  begin parallel computation
  - thread 1
    copy $\Omega_1(t)$ to $\Omega_1'(t)$
    apply $\Theta_1$ to $\Omega_1(t)$
  - thread 2
    copy $\Omega_2(t)$ to $\Omega_2'(t)$
    apply $\Theta_2$ to $\Omega_2(t)$ reading the remote context $V_{T_{21}}$ from $\Omega_1'(t)$.

  end parallel computation
- Copy the resulting $\Omega_2$ to $\Omega_1$, $t \rightarrow t+1$.

It is worth to be noted that the above algorithm is valid for both synchronous and asynchronous diffusion CA.

**Example 1.** A system of two interacting CA is used to simulate a pattern formation process [7] induced by a chemical reaction on a metallic surface, the latter having been heated in its central part. The CA $\aleph_v = \langle A_v, X_v, \Theta_v \rangle$ simulates the propagation of heat over the surface, being the asynchronous well known diffusion CA called a *naive diffusion* [6].
$\aleph_u = \langle A_u, X_u, \Theta_u \rangle$ simulates the process of patterns emergency on the surface. It is a synchronous CA of majority type with weighted template. The influence of changing temperature on pattern formation process is reflected by the dependence of weighted template entries on the averaged twin cells states of $\aleph_v$.

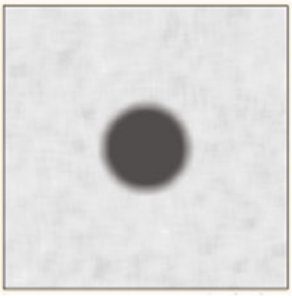


**Fig. 2.** Initial state of the cellular array $\Omega_v(0)$ from Example 1

Both CA have Boolean alphabet $A = \{0, 1\}$, their naming sets satisfying (1), $|X_v| = |X_u| = \{(i, j) : i, j = 0, \ldots, 300\}$.

CA $\aleph_v$ operates independently simulating the propagation of heat over the whole area, the initial distribution of temperature is shown in (Fig.2). According to naive CA-diffusion its local operator performs the exchange of states between a cell $((v_0, (i, j)_v)$ and one of its four neighbors $(v_k, (i, j)_v)$, $k = 1, 2, 3, 4$, using von Neumann template

$$T_v = \{(i, j)_v\}, \quad T''_{vv} = \{(i, j + 1)_v, (i - 1, j)_v, (i, j - 1)_v, (i + 1, j)_v\}, \tag{10}$$

and a transition function

$$\begin{aligned} v'_0 &= \quad v_k, \quad \text{if} \quad 0.25k < rand < 0.25(k + 1), \\ v'_k &= \begin{cases} v_0 & \text{if} \quad 0.25k < rand < 0.25(k + 1), \\ v_k & \text{otherwise.} \end{cases} \qquad k = 1, \ldots, 4. \end{aligned} \tag{11}$$

Local operator $\aleph_u$, operating in $\Omega_u = \{(u, (i, j)_u : (i, j) \in X_u\}$ has a single cell base, the context template including cells both from $X_v$ and $X_u$.

$$T''_{uv} = \{(i + g, j + h)_u, (i + g, j + h)_v : g, h = -3, \ldots, 3\}, \tag{12}$$

the transition function being as follows:

$$u'_0 = \begin{cases} 1, & \text{if} \quad \sum_{g=-r}^{r} \sum_{h=-r}^{r} w_{gh} u_{i+g,j+h} > 0.1, \\ 0, & \text{otherwise,} \end{cases} \tag{13}$$

where

$$w_{gh} = \begin{cases} 1, & \text{if} \quad |g| \leq 1 \ \& \ \ |h| \leq 1, \\ -\langle v_{i+g,j+h} \rangle & \text{otherwise.} \end{cases}$$

In Fig.3 three snapshots of pattern formation process in $\Omega_u$ are shown.

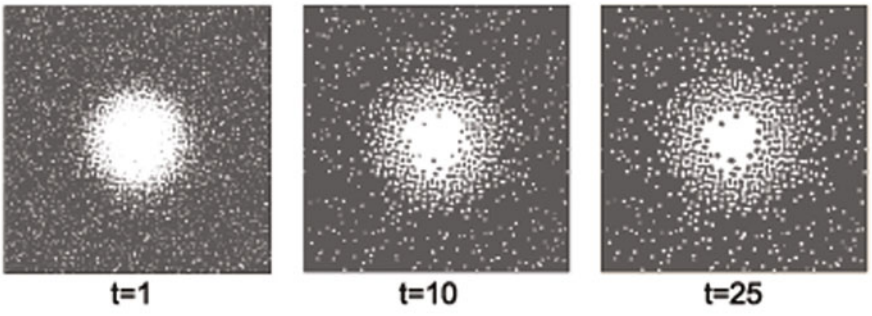



**Fig. 3.** Three snapshots of the evolution of a pattern formation CA $\aleph_u$ in a CA system where inhibitor values are controlled by the heat propagation $\aleph_v$ with initial state $\Omega_v(0)$ shown in Fig.2

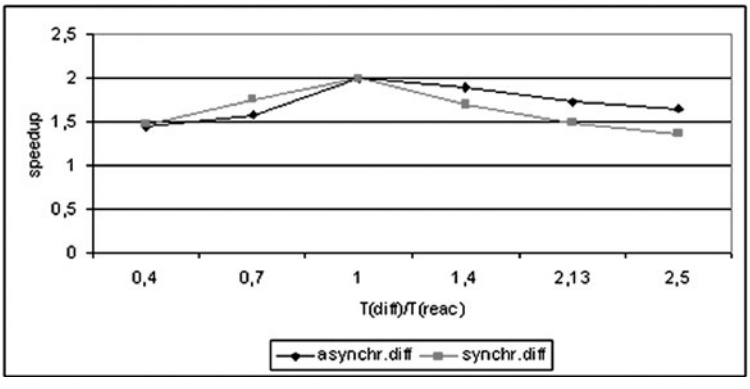



**Fig. 4.** Dependence of parallel two thread implementation efficiency on the ratio $\rho = T(diff)/T(react)$

The same process has been simulated by using diffusion CA $\aleph'_v$ of synchronous type with Margolus neighborhood [6]. The local operator of the latter is a superposition of two ones, $\Theta'_v = \Theta_1(\Theta_2)$ both being contextless, each having four cells in its base.

$$\begin{aligned}\Theta_1(i,j)_v = &\{((v_0,(i,j)_v),(v_1(i,j+1)_v),(v_2(i+1,j+1)_v),(v_3,(i+1,j)_v)\} \rightarrow \\ &\{((v'_0,(i,j)_v),(v'_1,i,j+1)_v),(v'_2(i+1,j+1)_v),(v'_3,(i+1,j)_v)\}, \\ \Theta_2(i,j)_v = &\{((z_0,(i,j)_v),(z_1(i,j+1)_v),(z_2(i+1,j+1)_v),(z_3,(i+1,j)_v)\} \rightarrow \\ &\{((z'_0,(i,j)_v),(z'_1,i,j+1)_v),(z'_2(i+1,j+1)_v),(z'_3,(i+1,j)_v)\},\end{aligned} \tag{14}$$

where

$$v'_k = v_{(k+1)mod4}, \quad z'_k = z_{(k-1)mod4}. \tag{15}$$

For both above CA systems, coordination of heat propagation rate and that of pattern formation is achieved by forming each iteration of the system by

including in it $Dv$ iterations of diffusion , and one iteration of pattern formation. The value of $Dv$ may vary from $Dv = 10$ to $Dv = 1000$. This fact gives us the opportunity to test efficiency of parallel implementation with different load balance between threads, by changing the value of $Dv$.

Computational experiments have been performed on the computer `Intel core i7` for two types of reaction-diffusion CA systems: 1) with asynchronous naive diffusion CA and 2) with Margolus CA-diffusion of synchronous type for five different values of $Dv$ in each case. The results of parallelization efficiency are given in Fig.4 in the form of speedup dependence on the ratio $\rho = T(diff)/T(react)$, where $T(diff)$ and $T(react)$ are diffusion and pattern formation one iteration computation times, respectively. It is seen that for both synchronous and asynchronous case the efficiency is perfect when $\rho = 1$. Also, it is high enough ($> 0,7$) with $0.7 < \rho < 1.3$.

## 5 CA System Simulating Many Interacting Processes

When many CA are functioning in common their interactions may be configured differently: some of them may operate independently, others may be interdependent. Correctness condition for any system is also expressed by (8) for each component CA and by (9) with the account that $T''(x_i)$ may be the union of several remote context templates. The most frequently studied is the type of CA system consisting of several reaction-diffusion blocks, for which two parallel implementation being possible:

1) each block is implemented as a single thread, reaction and diffusion CA running sequentially,

2) in each block reaction and diffusion are implemented as two parallel threads, hence, a system of $n$ interacting blocks requires $2n$ threads.

Of course, any intermediate case is possible, the best variant being when the computation load in all threads is close to be identical.

A typical example for testing multithread implementations of a complex reaction-diffusion system is a prey-predatory problem which is a well known one in mathematical ecology [8].

**Example 2.** Prey-predatory problem [8] is usually represented by a system of two PDEs.

$$\begin{aligned} u_{tt} &= d_u u_{xx} + F_u(u, v), \\ v_{tt} &= d_v v_{xx} + F_v(u, v), \end{aligned} \tag{16}$$

where $d_u, d_v$ are diffusion coefficients for two species, functions $F_u(u, v)$ and $F_v(u, v)$ are usually given in the form of polynomials of both variables. Let us interpret the problem in such a way: some predator (fish, deers) eat prey (plankton, moss). If there is enough of food, predator density increases (predator propagates) with the probability depending on satiated predator density. In case of food shortage predator density diminishes (predator die of hunger). Prey always

attempts to propagate, when not being eaten dy the predator. Since predator is usually more agile than prey, its diffusion is essential, as for prey diffusion – it is hardly observable, ($d_v >> d_u$).

CA system, representing prey-predatory interaction consists of two reaction-diffusion blocks: $\Upsilon_u$ and $\Upsilon_v$, each being a simple systems of two automata $\Upsilon_u = \langle \aleph_{u1}, \aleph_{u2} \rangle$, $\Upsilon_v = \langle \aleph_{v1}, \aleph_{v2} \rangle$, in both blocks the first CA simulates diffusion, the second – the reaction. Correspondingly, $\aleph_{u1} = \langle A_{u1}, X_{u1}, \Theta_{u1} \rangle$, $\aleph_{v1} = \langle A_{v1}, X_{v1}, \Theta_{v1} \rangle$.

All CA have Boolean alphabets. Any pair of $X_{lk} = \{(ij)_{lk}\}$, $(l = u, v)$, $k = (1, 2)$. meets the relation (1). In both blocks diffusion simulation is performed by using synchronous CA with local operators $\Theta_{v1}$ and $\Theta_{u1}$ given in Example 1 by (14), differing only in diffusion coefficients, which are in correspondence of prey and predator agility, expressed in the model by the number of iterations $D_v = 50$ or $D_u = 1$ to be performed during one iteration of the CA system.

Local operators of reaction CA $\Theta_{v2}$ and $\Theta_{u2}$ represent the behavior of predator and prey and depend on both local densities

$$V((i,j)_v) = \frac{1}{|Av(i,j)_v|} \sum_{(k,l) \in Av(i,j)_v} v(k,l)_v, \tag{17}$$

$$U((i,j)_u) = \frac{1}{|Av(i,j)_u|} \sum_{(k,l) \in Av(i,j)_u} u(k,l)_u,$$

where

$$Av(i,j) = \{(k,l) : \ k = i + g, l = j + h, \ \ g, j = -r, \ldots, r\},$$

$r = 8$ being the radius of averaging in both cellular arrays. For the predator local operator is as follows:

$$\Theta_{v2} : \ \{v(i,j)_{v2}\} \star V(T''((i,j)_{v2}) \rightarrow \{v'(i,j)_{v2}\} \tag{18}$$

where

$$T''(v(i,j)_{v2}) = \{(k,l)_{v1}, (k,l)_{u1} : k = (i+g)_{v1}, l = (j+h)_{v1} \\ g, j = -r, \ldots, r\} \tag{19}$$

and the next state value

$$v'((i,j)_{v2}) = \begin{cases} 0, & \text{if } \ \Delta V(i,j) < 0 \quad \& \quad (rand) < p_{v \rightarrow 0}, \\ 1, & \text{if } \ \Delta V(i,j) > 0 \quad \& \quad (rand) < p_{v \rightarrow 1}, \end{cases} \tag{20}$$

where

$$\Delta V(i,j) = V((i,j)_v) - U((i,j)_u).$$

The probabilities $p_{v \rightarrow 0}$ and $p_{v \rightarrow 1}$ are determined based on the following considerations:

- If $\Delta V(i,j) > 0$ , predator has lack of food and may die. So, $\Delta V((i,j)$ cell states in $Av((i,j)_{v2})$ should be inverted into "zero", yielding in $p_{v \rightarrow 0}$ be equal to the ratio $\Delta V(i,j)/U(i,j)$ [10].

- If $U(i,j) > V(i,j)$, $\Delta V((i,j)_{v2} < 0$, then predator has plenty of food at the place and propagates increasing its density according to the propagation function of the form $F_u(V) = cV(i,j)(1 - V(i,j)$, where $c = 0.5$ is a coefficient corresponding to the type of predator.

So, the probabilities in (20) are

$$\begin{array}{lll} p_{v\to 0} = \Delta V(i,j)/U(i,j), & \text{if} & \Delta V(i,j) > 0 \\ p_{v\to 1} = 0.5V(i,j)(1 - V(i,j)) & \text{if} & \Delta V(i,j) < 0. \end{array} \tag{21}$$

Next states in $\Theta_{u2}$ are computed similarly to those of $\Theta_{v2}$, the next-state functions being

$$u'((i,j)_{u2}) = \begin{cases} 0, & \text{if } \Delta U(i,j) < 0 \quad \& \quad (rand) < p_{u\to 0} \\ 1, & \text{if } \Delta U(i,j) > 0 \quad \& \quad (rand) < p_{u\to 1} \end{cases} \tag{22}$$

where

$$\Delta U(i,j) = U(i,j) - V(i,j)$$

probability values $p_{u\to 0}$ and $p_{u\to 1}$ in (22) are based on the following considerations:

- If $\Delta U(i,j) > 0$, prey is freely eaten. So, its density decreases with probability proportional to predatory density.
- If $\Delta U(i,j) < 0$, prey propagates with probability proportional to the number of the remainders

$$\begin{array}{lll} p_{u\to 0} = \Delta U(i,j) & \text{if} & \Delta U(i,j) > 0 \\ p_{u\to 1} = 0.5(\Delta U(i,j)(1 - \Delta U(i,j)) & \text{if} & \Delta U(i,j) > 0. \end{array} \tag{23}$$

Let $X = \{(i,j) : i = 0,\ldots,399, /j = 0,\ldots,799\}$. In the initial state, prey is spread over $\Omega_{u1}(0)$ with density, $U((i,j)_{u1}(0) = 0.4$ for all $(i,j)_{u1}) \in X_{u1}$. Predator has the density $V((i,j)_{v1}) = 0.1$ for the whole $\Omega_{v1}(0)$ except a band $\{(i,j)_{v1} : i = 0,\ldots,399, j = 369,\ldots,439\}$ where $V((i,j)_{v1}) = 1$ (Fig.5, t=0). Each $t$th iteration of the CA system consists of four following parts:

- V-Diffusion:
  1. $\Omega_{v1}(t)$ is copied to $\Omega'_{v1}(t)$.
  2. $D_v$ iterations of $\aleph_{v1}$ are performed by application of $\Theta_{v1}$ to transfer from $\Omega_{v1}(t)$ to $\Omega_{v1}(t+1)$.
- U-Diffusion:
  1. $\Omega_{u1}(t)$ is copied to $\Omega'_{u1}(t)$.
  2. $D_u$ iterations of $\aleph_{u1}$ are performed by application of $\Theta_{v1}$ to transfer $\Omega_{u1}(t)$ to $\Omega_{u1}(t+1)$.
- V-reaction:
  An iteration of $\aleph_{v2}$ is performed by applying $\Theta_{v2}$ (18) to all $(i,j)_{v2} \in X_{v2}$, probabilities being computed by (21), the resulting $\Omega_{v2}$ is copied to $\Omega_{v1}$.
- U-reaction:
  An iteration of $\aleph_{u2}$ is performed by applying $\Theta_{u2}$ (22) to all $(i,j)_{u2} \in X_{v2}$, probabilities being computed from (23), the resulting $\Omega_{u2}$ is copied to $\Omega_{u1}$.

In Fig. 5 four snapshots of the evolution of the predator component are shown, obtained by sequential implementation of the above four parts of the whole algorithm. CA system comes to its stable state rather slowly: the snapshot (t=500) is not yet close to it, but just that iteration number has been used for comparing times of sequential and parallel implementation.

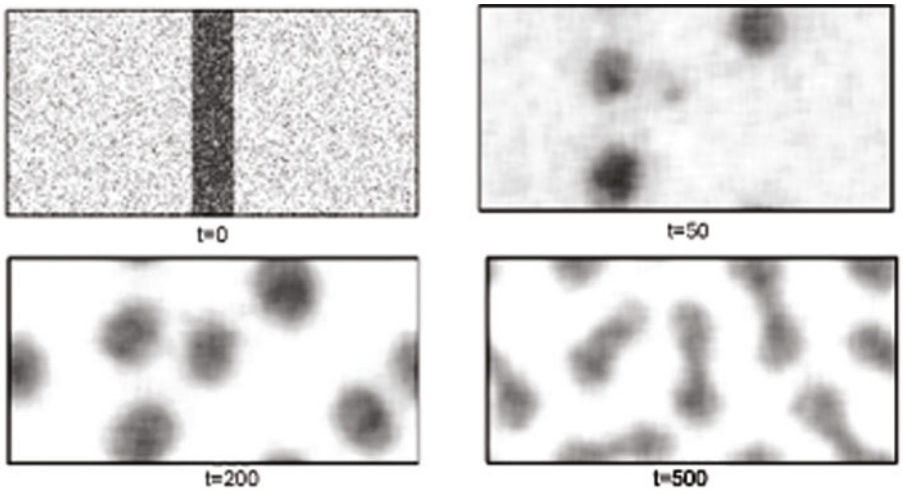



**Fig. 5.** Four snapshots of the evolution of predatory CA $\aleph_{v2}$ in prey-predatory CA system : initial cellular array $\Omega(0)$ in Boolean form, $\Omega(50)$, $\Omega(200)$ and $\Omega(500)$ – in averaged form

**Table 1.** Computation time of 500 iterations and speedup of 1, 2, and 4 thread implementation of the CA -system of Example 2

| number of threads | 1 | | 2 | | 4 | |
|---|---|---|---|---|---|---|
| | time | speedup | time | speedup | time | speedup |
| CA, Dv/Du=50 | 134 | 1 | 77 | 1.74 | 47 | 2.35 |
| CA, Dv/Du=1 | 150 | 1 | 77 | 1.94 | 57 | 2.63 |
| Dom-decomp | 134 | 1 | 72 | 1.85 | 39 | 3.4 |

Implementation of the above CA system in two threads has been made by combining operation of $\aleph_{v1}$ and $\aleph_{v2}$ in one thread, and $\aleph_{u1}$ and $\aleph_{u2}$ — in the other thread. Implementation of the system in four threads has been performed by allocating each part of the above algorithm onto a thread. Due to the difference of the diffusion coefficients, the computation load in the threads differs essentially, which does not allow to obtain the perfect speedup. Implementation of a CA-system with equal thread load ($Dv = Du = 50$) results in the speedup is close to the number of threads. The obtained results with $T = 500$ iterations

are compared to those obtained by the domain decomposition method, when each thread processes its own parts of two interacting cellular arrays, the thread loads being identical. The results of the above experiments are summarized in Table 1. It is seen from the table, that the speedup does not depend on the parallelization method, only the imbalance of thread load is essential.

## 6 Conclusion

A concept of CA system is introduced, which is a set of CA working in common sharing common variables. Parallelization method for implementing CA-systems in multicore computer is presented. The method is based on associating each CA of the system to a thread of the parallel algorithm. Computational experiments were performed by simulating evolution of reaction-diffusion systems on a multicore computer of the type Intel i7. They show, that the speedup depends on the imbalance of the load between threads. It is equal to $n$ (the number of cores used) when computational load in threads is identical, and quite acceptable ($> 0.8n$) when the load ratio is $0.8 < \rho < 1.2$. Comparison of obtained efficiency with that of parallelization using domain decomposition resulted in minor difference in the above interval of load imbalance.

## References

1. Wolfram, S.: Cellular Automata abd Comlexity – Collected paopers. Addison Wesley, Reading (1994)
2. Hoekstra, A.G., Kroc, J., Sloot, P.M.A. (eds.): Simulating Complex Systems by Cellular Automata. Understanding complex Systems. Springer, Berlin (2010)
3. Bandman, O.: Cellular Automarta Composition Techniques for Spatial Automata Simulation. In: Hoekstra, A.G., Kroc, J., Sloot, P.M.A. (eds.) Simulating Complex Systems by Cellular Automata. Understanding complex Systems, pp. 81–115. Springer, Berlin (2010)
4. Achasova, S., Bandman, O., Markova, V., Piskunov, S.: Parallel Substitution Algorithm. Theory and Application. World Scientific, Singapore (1994)
5. Bandman, O.: Coarse-Grained Parallelization of Cellular-Automata Simulation Algorithms. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 370–384. Springer, Heidelberg (2007)
6. Toffolli, T., Margolus, N.: Cellular Automata Machine. MIT Press, USA (1987)
7. Deutsch, A., Dorman, S.: Cellular Automata Modeling of Biological Pattern Formation. Birkhäuser, Berlin (2005)
8. Cataneo, G., Dennunzio, A., Farina, F.: A Full Cellular Automaton to Simulate Predatory-Prey Systems. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 446–451. Springer, Heidelberg (2006)
9. Chua, L.: CNN: a paradigm of complexity. World Scientific, Singapore (2002)
10. Bandman, O.: Simulating Spatial Dynamics by Pribabilistic Cellular Automata. In: Bandini, S., Chopard, B., Tomassini, M. (eds.) ACRI 2002. LNCS, vol. 2493, pp. 10–16. Springer, Heidelberg (2002)

# Efficient Minimal Routing in the Triangular Grid with Six Channels

Rolf Hoffmann[1] and Dominique Désérable[2]

[1] Technische Universität Darmstadt, FB Informatik, FG Rechnerarchitektur, Hochschulstraße 10, 64289 Darmstadt, Germany
[2] Laboratoire LGCGM UPRES EA 3913, Institut National des Sciences Appliquées, 20 Avenue des Buttes de Coësmes, 35043 Rennes, France
hoffmann@ra.informatik.tu-darmstadt.de,
deserabl@insa-rennes.fr

**Abstract.** This paper describes an efficient novel router on the 6-valent triangular grid with toroidal connections, denoted "$T$–grid" in the sequel. The router uses six channels per node that can host up to six agents. The topological properties of the $T$–grid are given first, as well as a minimal routing scheme, as a basis for a Cellular Automata modeling of this new target searching problem. Each agent situated on a channel has a computed "minimal" direction defining the new channel in the adjacent node. When moving to the next node an agent can simultaneously hop to another channel. In the normal protocol the rightmost subpath (from the agent's point of view) is taken first. In addition, an adaptive routing protocol is defined, preferring the direction to an unoccupied channel. The novel router is significantly faster than an optimized reference router with only one agent per node that was designed before. In order to avoid deadlocks, the initial setting of the channels are alternated in space.

**Keywords:** Routing, Triangular Torus, Cellular Automata (CA), Multi-Agent System.

## 1 Introduction

In order to communicate between processors on a chip an appropriate network has to be supplied. A lot of research has been carried out in order to find the best networks with respect to latency, throughput, fault tolerance, and so on. Instead of improving the known design principles we follow the approach based on agents transporting messages from a source node to a destination node. Here we will focus on the case where the agents have to follow a minimal route (or shortest path). The underlying network we are investigating is the *triangular torus* with cyclic connections, denoted "$T$–grid" in the sequel. The nodes are connected via twelve unidirectional links, namely two in each of the six directions, that corresponds with a full-duplex or double lane traffic. Each node is provided with six channels, also called *cells* or *buffers* depending on the context. Each channel may host an agent in order to transport a message.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 152–165, 2011.

This paper follows a previous work dealing with a similar task running on the $T$–grid: see [1] and references therein. The novelty is that six agents per node are now used, with one agent per channel, instead of one agent per node therein. Here each agent moves to the next node, defined by the channel's position it is situated on. When moving to the next node, an agent may hop to another channel, defined by the direction of the agent. An agent can choose its new direction only among the directions that belong to the minimal path, namely the subpaths of the "minimal" parallelogram between source and destination, as shown in Fig. 2.

Another difference is that in [1] the agent's behavior is controlled by a finite state machine (FSM) evolved by a genetic algorithm, whereas here the behavior is handcrafted. The peculiar interest of this study is that it yields comparative results on the performance of routing protocols between two networks with a different number of buffers per node (6 vs 1). Thus the goal of this paper is to find a faster router on the $T$–grid using six agents per node and bidirectional traffic between nodes, at first modeling the system as Cellular Automata (CA) and then discussing whether the routing algorithm is deadlock-free or not.

**Related Work.** Target searching has been researched in many variations: with moving targets [2] or as single-agent systems [3]. Here we restrict our investigation to stationary targets, and multiple agents having only a local view. This contribution continues our preceding work on routing with agents on the cyclic triangular grid [1] used for comparison and on non-cyclic rectangular 2D mesh grids [4]. In order to get a deadlock-free algorithm, it is necessary to add a small amount of randomness to a deterministic behavior [5]. In a recent work [6], the $T$–grid and the square grid (or "$S$–grid"), both with cyclic connections, were compared. Evolved agents, with a maximum of one agent per cell, were used in both cases. It turned out that the $T$–grid performed significantly better than the $S$–grid.

The remainder of this paper is structured as follows. Section 2 deals with the topology of the $T$–grid and presents a shortcut of the routing scheme within it. The framework for the agents' routing task is defined in Sect. 3. Section 4 shows how the routing can by modeled as a multi-agent system in the CA network. An analysis of the router efficiency is discussed in Sect. 5 and some deadlock situations are pointed out before Conclusion.

## 2 Topology and Routing in the CA Network

### 2.1 Topology of the $T$–grid

Consider the square blocks in Fig. 1 with $N = 2^n \times 2^n$ nodes where $n$ will denote the "size" of the networks. The nodes are labeled according to the XY–orthogonal coordinate system. In the left block, a node $(x, y)$ labeled "$xy$" is connected with its four neighbors $(x \pm 1, y)$, $(x, y \pm 1)$ (with addition modulo $2^n$) respectively in the $N$–$S$, $W$–$E$ directions, giving a 4–valent torus usually denoted as "square" and labeled "S" or "$S$–grid" elsewhere [6]. In the right block, two additional links $(x - 1, y - 1)$, $(x + 1, y + 1)$ are provided in the diagonal $NW$–$SE$ direction,

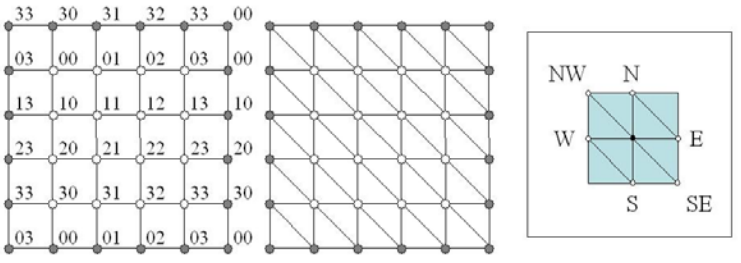



**Fig. 1.** Tori "$S$" and "$T$" of size $n = 2$, of order $N = 16$, labeled in the $XY$ coordinate system; redundant nodes in grey on the boundary. Orientations $N$–$S$, $W$–$E$, $NW$–$SE$ in the inset, according to an $XYZ$ reference frame (not displayed).

giving a 6–valent torus usually denoted as "triangulate" and labeled "T" or "$T$–grid" in the sequel. Because their associated graphs are regular their number of links is, respectively, $2N$ for torus $S$ and $3N$ for torus $T$. Both networks are scalable in the sense that one network of size $n$ can be built from four blocks of size $n-1$. The $S$–grid is just displayed here because it is often interesting to compare the topologies and performances of $S$ and $T$, two networks of the same size; moreover, $S$ can be somehow viewed as a subgrid of $T$.

As a matter of fact, the $T$–grid belongs to a family of hierarchical Cayley graphs generated in the hexavalent grid. The associated dual tessellation of the plane is the regular hexagonal tiling, or is homeomorphic to, and it is well known that this "honeycomb" or honeycomb-like tiling is provided with the maximum of symmetries. The graphs of this family are denoted elsewhere as "arrowhead" or "diamond" in order to avoid confusion with other families of hexavalent networks [7]. The reader is referred to [8] for more details about the genesis of these graphs and some of their topological properties. It was also shown that these graphs provide a good framework for routing [9] and other global communications like broadcasting [10] and gossiping [11]. A very important property is that as Cayley graphs they are vertex-transitive, that means that any vertex behaves identically.

An important parameter for the routing task in the networks is the diameter. The diameter defines the shortest distance between the most distant pair of nodes and provides a lower bound for routing or other global communications; such a pair is said to be antipodal. The diameters $D_n$ in $S$ and $T$ are given by

$$D_n^S = \sqrt{N}; \qquad D_n^T = \frac{2(\sqrt{N}-1)+\varepsilon_n}{3}$$

where $\varepsilon_n = 1$ (resp. 0) depends on the odd (resp. even) parity of $n$ and where the upper symbol identifies the torus type; whence the ratio denoted

$$D_n^{S/T} \approx 1.5$$

between diameters. In this study, only the diameter $D_n^T$ will be considered, denoted simply $D_n$ in the sequel [12].

### 2.2 Routing Scheme in the $T$–Grid

The basic routing schemes are driven by the Manhattan distance in the 4–valent square grid [13] and by the so-called "hexagonal" distance in the 6–valent triangular grid [9]. They are denoted as "rectangular" and "triangular" herein. Considering a source "$S$" and a target "$T$" as shown in Fig. 2, we choose to find a shortest path from $S$ to $T$ with *at most one change* of direction. It would be convenient to attach a system of axes to both $S$ and $T$.

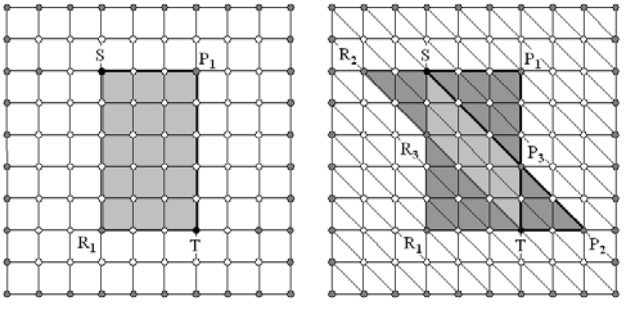



**Fig. 2.** Networks of size $n = 3$, of order $N = 64$. Routing paths from a source "S" to a target "T": rectangular routing in the 4–valent grid on the left, triangular routing in the 6–valent grid on the right. In the rectangular routing, axis systems $X_S Y_S$ and $X_T Y_T$ intersect at $P_1, R_1$ and yields the rectangle $SP_1TR_1$ in general. In the triangular routing, axis systems $X_S Y_S Z_S$ and $X_T Y_T Z_T$ intersect at $P_i, R_i$ $(i = 1, 2, 3)$ and yields three parallelograms $SP_iTR_i$ in general; in this case, the parallelogram $SP_3TR_3$ is "minimal".

In the square grid on the left part, the construction yields the rectangle $SP_1TR_1$. In order to ensure a homogeneous routing scheme, from an usual convention the agent is carried following line $X$ first, following line $Y$ afterwards. Under these conditions, a route $S \rightarrow T$ and a route $T \rightarrow S$ will follow two disjoint paths and each of them is made of two unidirectional subpaths, that is $S \rightarrow P_1 \rightarrow T$ and $T \rightarrow R_1 \rightarrow S$ respectively. In a particular case, $S$ and $T$ may share a common axis and the routes $S \rightarrow T$ and $T \rightarrow S$ need a (full-duplex) two-lane way $S \leftrightarrow T$. Note that in a finite-sized torus, not only the "geometric" rectangle $SP_1TR_1$ should be considered but rather a "generalized" rectangle, because the unidirectional subpaths may "cross" over the boundaries of the torus: a shrewd reader could check that it is the case on this example in Fig. 2.

In the triangular grid on the right part, the construction is somewhat more tricky and involves three generalized parallelograms of the form $SP_iTR_i$. Among them, there exists a "minimal" one that defines the shortest path. It is the purpose of this paper to detect it and to move CA mobile agents within it.

## 3 The Agents' Routing Task

Considered is the $T$–grid of $N$ nodes as described in the previous section. Each node contains six channels denoted in the sequel by *cells*, or *buffers*, according to the context. A node acts as a communication node or a processor in the network.

A *message transfer* is the transfer of one message from a source to a target, each agent shall perform such a message transfer. A set of messages to be transported is called *message set*. A *message set transfer* is the successful transfer of all messages belonging to the set. Initially $k$ agents are situated at their source nodes. Then they move to their target nodes on certain channels. When an agent reaches its target, it is deleted. Thereby the number of moving agents is reduced until no agent is left. This event defines the end of the whole message set transfer. Note that the agents hinder each other more at the beginning (due to congestion) and less when many of the agents have reached their targets and have been deleted. No new messages are inserted into the system until all messages of the current set have reached their targets. This corresponds to a barrier-synchronization between successive sets of messages. Initially each agent is placed on a certain channel (with direction to the target) in the source node and each agent knows its target. The target node of an agent should not be its source node: message transfers within a node without an agent's movement are not allowed. Two test cases will be used for evaluation, where $k$ is the number of agents, $s$ the number of source nodes and $d$ the number of target nodes:

1. **First Test Case.** ($d = 1, k = s$) All agents move to the same common target. We will consider the case $k = N-1$, meaning that initially an agent is placed on each site (except on the target). In this case the optimal performance of the network would be reached if the target consumes six messages in every timestep ($t = (N-1)/6$). In addition, the target location is varying, with a maximum of $N$ test configurations in order to check the routing scheme exhaustively. We recall that the $T$–grid is vertex-transitive, so the induced routing algorithm must yield the same result for all $N$ cases!
2. **Second Test Case.** ($k = s = d$) The sources are mutually exclusive (each source is used only once in a message set) and targets are mutually exclusive (each target is used only once). Source locations may act as targets for other agents, too. We will consider the case $k = N/2$ that was also used in preceding works [1,6] for comparison. Note that the minimum number of timesteps $t$ to fulfil the task is the longest distance between source and target which is contained in the message set. For a high initial density of agents the probability is high that the longest distance is close to the diameter of the network. Thus the best case would be $t = D_n$.

The goal is to find an agent's behavior in order to transfer a message set (averaged over many different sets) as fast as possible, that is, within a minimal number of generations $t$. We know from previous works that the agent's behavior can be optimized (e.g. by a genetic algorithm) with respect to the set of given initial configurations, the initial density of agents, and the size of the network. The goal here is not to fully optimize the agent's behavior but rather to design

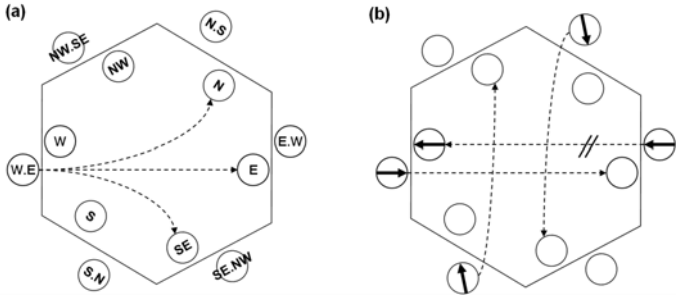



**Fig. 3.** (a) Each hexagonal node contains six channels (buffers, cells) that can hold agents (messages). The channels denoted $N$, $E$, $SE$, $S$, $W$, $NW$ transport messages to the adjacent node in front. For example, an agent situated on the $E$–channel of the western cell "$W.E$" can move to $E$ or $N$ or $SE$.– (b) Agents have computed directions, depicted as encircled arrows. Agent at $W.E$ moves to $E$. Agent at $E.W$ cannot move because $W$ is occupied. Agent at $N.S$ with direction `toSE` moves to $SE$. Agent at $S.N$ with direction `toNW` moves to $NW$.

a good agent system with six channels that outperforms the results carried out in [1,6].

## 4 CA Modeling of the Multi-agent System

### 4.1 Cellular Automata Modeling

The whole system is modeled as a CA where $N = 2^n \times 2^n = M \times M$ nodes are arranged as in the $T$-grid of Fig. 1. Each node is labeled by its $(x, y)$ coordinates defining the node's site. The node contains six cells, identified by the channels

$$C_i \in \{C_0, C_1, C_2, C_3, C_4, C_5\} = \{E, SE, S, W, NW, N\}$$

labeled clockwise as shown in Fig. 4c. Index $i$ is also called *position* or *lane number* in this context. The position of a channel defines also an implicit direction / orientation that defines the next *adjacent* node that an agent visits next on its travel. Each agent has a direction which is updated when it moves. In the general case, the direction is one out of two possibilities defined by both unidirectional subpaths of the minimal route; otherwise, agent and target lie on the same axis and there is only one unidirectional path.

An agent can move to one out of three channels of the adjacent node provided by the hardware, as shown in Fig. 3a. For example, an agent can move from node $W$ at $(x-1, y)$, channel $W.E$, to node at $(x, y)$, channel $E$ or $N$ or $SE$. The notation "$W.E$" stands for the $E$–channel of the $W$–neighbor. In other words, the current direction of the agent defines the channel in the next node where the agent will move to. A receiving channel may solve a conflict if there are several sending channels (Fig. 3b).

The cell's state is given by

$$C = (c, (x', y'))$$

where $(x', y')$ denotes the target coordinates, $c \in D$ stands for the direction of the agent and

$$D = \{0, 1, 2, 3, 4, 5\} \equiv (\texttt{toE}, \texttt{toNW}, \texttt{toS}, \texttt{toW}, \texttt{toSE}, \texttt{toN})$$

whereas the empty cell is encoded by $c = -1 = \omega$. In a graphical representation, the directions can be symbolized by $(\rightarrow, \nwarrow, \downarrow, \leftarrow, \searrow, \uparrow)$ according to the inset in Fig. 1. In addition, the six direct neighboring cells are denoted by $M_{j(i)}$ as displayed in Fig. 4c, indexed relatively to channel $i$. The three channels opposite to channel $C_i$ are also denoted by

$$R_i = M_{i+4}(\texttt{fromRight}), \quad S_i = M_{i+3}(\texttt{fromStraight}), \quad L_i = M_{i+2}(\texttt{fromLeft}).$$

These channels are of special interest because the agents may only move from $R_i$, $S_i$ or $L_i$ to $C_i$ on their minimal route. Therefore $R_i$, $S_i$ and $L_i$ are "copy"–neighbors of $C_i$ that need to be checked in order to copy an agent (Fig. 4a). In order to delete an agent on $S_i$ for instance, the *move-to* conditions of $C_i, C_{i-1}, C_{i+1}$ have also to be evaluated by $S_i$ and therefore an extended "delete"–neighborhood

$$(C_i, C_{i-1}, C_{i+1}, L_i, R_i, LL_i, RR_i)$$

is necessary in the CA model (Fig. 4b). Note that the cardinality of the neighborhood of a receiving cell is 3 whereas it is 7 for a sending cell, without counting the own cell. As a matter of fact, the delete–neighborhood is dynamic: depending on the direction, only three neighbors need to be checked: e.g., the actual neighbors are $NW.SE$, $S.N$, $E$ if the agent wants to move straight from $W.E$ to $E$. The whole neighborhood in the CA model is the union of the copy- and delete–neighborhood, namely 3 channels from the left and 7 channels to the right with respect to cell $E$.

At first each channel $C_i$ computes several conditions defining which of the incoming agents will be hosted next:

- Agent wants to move from $L$ to $C$, first priority: $\texttt{LtoC} = (l = i)$
- Agent wants to move from $S$ to $C$, second priority: $\texttt{StoC} = (s = i) \wedge \neg\texttt{LtoC}$
- Agent wants to move from $R$ to $C$, third priority: $\texttt{RtoC} = (r = i) \wedge \neg\texttt{RtoC}$.

Using these conditions, five cases are distinguished:

- (case $\kappa_1$) : $(c \neq \omega)$ // cell not empty, agent stays at rest
- (case $\kappa_2$) : $(c = \omega) \wedge \texttt{LtoC}$ // cell empty, agent to be copied from L
- (case $\kappa_3$) : $(c = \omega) \wedge \texttt{StoC}$ // cell empty, agent to be copied from S
- (case $\kappa_4$) : $(c = \omega) \wedge \texttt{RtoC}$ // cell empty, agent to be copied from R
- (case $\kappa_5$) : $(c = \omega) \wedge \neg\texttt{LtoC} \wedge \neg\texttt{StoC} \wedge \neg\texttt{RtoC}$ // cell remains empty

Then the target coordinates $(x', y')^*$ are copied from L, S, or R if the agent moves to C, as

$$(x', y')^* = (x', y')[L] \quad \text{IF} \quad \kappa_2$$
$$(x', y')^* = (x', y')[S] \quad \text{IF} \quad \kappa_3$$
$$(x', y')^* = (x', y')[R] \quad \text{IF} \quad \kappa_4$$

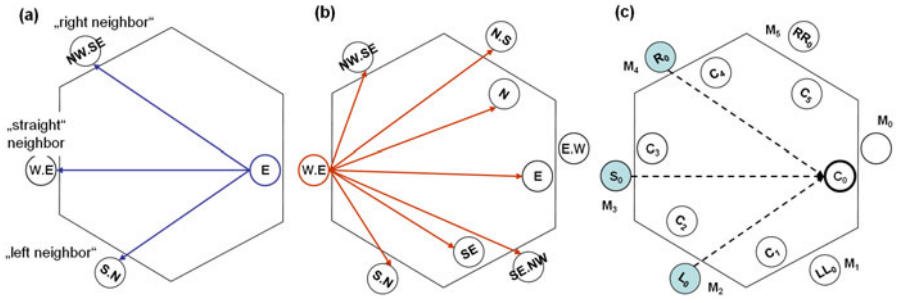



**Fig. 4.** Neighborhood. – (a) Cell $E$ has three "copy"–neighbors: right neighbor, straight neighbor and left neighbor. The same "relative" neighborhood is valid for the other five cells because of the rotational invariance. – (b) In order to delete an agent on the own cell in case of movement, the own cell must evaluate the rules of all possible receiving cells. The possible receiving cells for $W.E$ are $E$, $N$, $SE$. Thus not only $E$, $N$, $SE$ are the "delete"–neighbors of $W.E$, but also the "copy"–neighbors of them, i.e. $NW.SE$, $S.N$, $N.S$, $SE.NW$. – (c) Labeling of the channels used for the description of the CA rule, with respect to the receiving channel $C_i = C_0$. Channels $M_i$ are located in adjacent nodes. The sending channels with respect to $C_0$ are $R_0$, $S_0$, $L_0$.

then the new direction is computed using the function

$$\varphi : c^* = \varphi((x, y), (x', y')^*)$$

and finally, the agent's direction is updated synchronously:

$$c \leftarrow c^* \quad \text{IF} \quad \kappa_{i \in 1,2,3,4}.$$

For case $\kappa_5$, the direction is irrelevant and needs not to be updated.

If an agent moves, it is deleted on its old site. The delete–part of the CA rule is not given here and it is complicated because of the extended neighborhood. In the software simulation program, the GCA–w model with local write access to the neighbor was used [14]. This model allows to write onto a neighbor. It is especially useful if there are no write conflicts (exclusive-write condition), which here is the case, because an agent is copied exactly by one receiving cell, and then the receiving cell can at the same time delete the agent on the sending cell.

It is assumed that the agents are initially placed on a channel which is part of the minimal route, and the initial direction is one of the minimal directions.

Function $\varphi$ computes one of the directions which is minimal with respect to the target. If the target is straight ahead (or behind), then there is only one unique direction and the agent has to move straight ($\texttt{dirR} = \texttt{dirL} = 0°$). If the target lies within an angle $\alpha$ of ($-60° < \alpha < 0°$, degrees counted clockwise) relatively to the orientation of the channel in the node on which the agent is placed, then there are two minimal alternatives: take the "left" choice ($\texttt{dirL} = -60°$), or take the "right" choice ($\texttt{dirR} = 0°$). If the target lies within an angle $\alpha$ of ($+60° > \alpha > 0°$) then the "left" choice is ($\texttt{dirL} = 0°$) and the "right" choice is ($\texttt{dirR} = +60°$).

### 4.2 Computing the Minimal Route

The following abbreviations are used in the routing algorithm:

$sign(d) = (0, 1, -1)$ IF $(d = 0, d > 0, d < 0)$

$\bar{d} = d - sign(d) \cdot M/2$, where $M = 2^n$ is the length of any unidirectional cycle.

STEP 0. The deltas between the target and the actual position are computed.
$(dx, dy) := (x'^* - x, y'^* - y)$.

STEP 1. The deltas are contracted to the interval $[-M/2, +M/2]$.
$dx := \bar{dx}$ IF $|dx| > M/2$ ; $dy := \bar{dy}$ IF $|dy| > M/2$

If $sign(dx) = sign(dy)$ then the minimal path is already determined and the diagonal is used as one of the subpaths. Note that the path length is given by $max(|dx|, |dy|)$ if the signs are equal, by $|dx| + |dy|$ otherwise.

STEP 2. One of the following operations is performed, only if $dx \cdot dy < 0$. They comprise a test whether the path with or without using the diagonal is shorter.
$dx := \bar{dx}$ IF $|dx| > |dy|$ AND $|\bar{dx}| < |dx| + |dy|$ // $|\bar{dx}| = max(|dx|, |dy|)$
$dy := \bar{dy}$ IF $|dy| \geq |dx|$ AND $|\bar{dy}| < |dx| + |dy|$ // $|\bar{dy}| = max(|dx|, |dy|)$

STEP 3. This step forces the agents to move in the same direction if source and target lie opposite to each other, namely at distance $M/2$ on the same axis. Thereby collisions on a common node on inverse routes are avoided.

$(dx, dy) := (|dx|, |dy|)$ IF $(dx = -M/2)$ AND $(dy = -M/2)$
$dx := |dx|$ IF $(dx = -M/2)$ AND $(dy = 0)$
$dy := |dy|$ IF $(dy = -M/2)$ AND $(dx = 0)$

Then a minimal route is computed as follows:

(a) If $dx \cdot dy < 0$ then FIRST move $dx' = dx$ steps, THEN move $dy' = dy$ steps
(b) If $dx \cdot dy > 0$ then calculate
  (1) $dz' = sign(dx) \cdot min(|dx|, |dy|)$ // steps on the diagonal
  (2) $dx' = dx - dz'$, $dy' = dy - dz'$

Move FIRST $dz'$ THEN $dx'$ (IF $dy' = 0$) or move FIRST $dy'$ THEN $dz'$ (IF $dx' = 0$).

This algorithm yields a minimal route and uses a cyclic priority for the six directions, two or one of them which are used in a valid minimal route. For short, the algorithm uses the priority scheme:

FIRST $dx'$ THEN $dy'$ (IF $dz' = 0$),
FIRST $dz'$ THEN $dx'$ (IF $dy' = 0$),
FIRST $dy'$ THEN $dz'$ (IF $dx' = 0$).

This priority scheme means use "FIRST `dirR` and THEN `dirL`". An equivalent symmetric protocol would be "FIRST `dirL` THEN `dirR`".

Finally, the function $\varphi$ is applied. It has the task to select dynamically one of the minimal directions, either `dirR` or `dirL`. Different functions were tested in order to achieve the best performance or to avoid deadlocks. The following functions are used here:

1. (a) Initially the direction $\varphi =$ `dirL` is used.
   (b) Initially $\varphi =$ `dirL` IF $x + y \equiv 0$ and $\varphi =$ `dirR` IF $x + y \equiv 1 \mod 2$.
2. During the run, the direction $\varphi =$ `dirL` is always taken, meaning that the agents always prefer the "left" choice.
3. During the run, if the temporary computed direction (e.g., `dirL`) points to an occupied channel, then the other channel (e.g., `dirR`) is selected no matter this channel is free or not. This technique will be called *adaptive* routing.

The adaptive technique was manually designed and is a simple algorithm defining the new direction of the agent. It was not the purpose of this paper to find the best agent's behavior which is subject for further research. It should be noted that the CA herein can be seen as a "partitioned CA" [15]. Another way of modeling such a system would be to use a hexavalent FHP–like Lattice Gas CA [16]; but here the purpose is to avoid the two-stage timestep in order to save time, with only one clock cycle instead of two.

## 5 Router Efficiency and Deadlocks

### 5.1 Efficiency of Non-adaptive Routing

Using one agent only in the router, it will travel always on a minimal route. More agents are also using a minimal route, but sometimes they have to wait due to traffic congestion.

**First Test Case.** In the first test case scenario, $k = N - 1$ messages move to the same common target from all other nodes. All possible or a large number of initial configurations differing in the target location were tested (Tab. 1). The results are the same for all tested initial configurations. This means that the router works totally symmetric as expected. An optimal router would consume in every generation six agents at the target, leading to an optimum of $t_{opt} = k/6$. It is difficult to reach the optimum, because the agents would need a global or a far view in order to let the agents move simultaneously in a cohort. Here an agent needs an empty receiver channel in front in order to move, thus empty channels are necessary to signal to the agents that they can move.

As the router is completely filled with agents at the beginning (one agent in each node except the target node), there exist some agents which have as travel distance the diameter $D_n$. Therefore the ratio $t/D_n$ ($B/C$ in Tab. 1) is significantly higher than one, slightly higher than $M/2$. On the other hand, the ratio $t/(k/6) = B/D$ is quite good and relatively constant, that is $B/D \approx 2$ for large $N$, which is almost optimal because each agent needs an empty channel in front when moving without deviation on the minimal route.

**Second Test Case.** This test case was already used in a previous work [6] and is used for comparison. Therein, the agents were controlled by a finite state

**Table 1.** Message transfer time (in *timesteps*) of the $T$–grid, averaged over the number of checked initial configurations. First test case scenario: $k = N - 1$ messages travel from all disjoint sources to the same common target. The time is independent of the position of the target.

| Nodes N | Number of configurations (destinations) checked | (B) Message transfer time [steps] | (C) Diameter | Ratio B/C | (D) N/6 | Ratio B/D |
|---|---|---|---|---|---|---|
| 4=2 x 2 | all 4 | **1** | 1 | **1** | 1 | **1** |
| 16=4 x 4 | all 16 | **5** | 2 | **2.5** | 3 | **1.67** |
| 64=8 x 8 | all 64 | **23** | 5 | **4.6** | 11 | **2.09** |
| 256=16 x 16 | all 256 | **89** | 10 | **8.9** | 43 | **2.07** |
| 1024=32 x 32 | all 1024 | **351** | 21 | **16.7** | 171 | **2.05** |
| 4096=64 x 64 | 64 | **1384** | 42 | **32.95** | 683 | **2.03** |

machine FSM: optimized, evolved agents were used, choosing a random direction with probability 0.3% in order to avoid deadlocks, with only one agent per node. Ratio $A/B$ in Tab. 2 shows that even the non-adaptive router with six channels performs significantly better, with $A/B \approx 2.5$ for $N = 1024$. The main reason is that here a node can host six agents, not only one, and therefore the congestion is lower. Thus the main goal of this work is reached, namely to find a more efficient router in the $T$–grid with six channels.

**Table 2.** Message transfer time for $N/2$ messages (second test case). Sources are disjoint, and destinations are disjoint. Routing with six channels per node performs significantly better (ratio A/B) than FSM controlled agents (one per node).

| Nodes N | Number of configurations checked for B randomly generated | (A) time steps, FSM controlled agent | (B) Time steps, 6 channels non-adaptive | Ratio A/B | (C) Diameter | Ratio B/C | Time steps, 6 channels adaptive |
|---|---|---|---|---|---|---|---|
| 4=2 x 2 | 32 | 3.756 | 1 | **3.76** | 1 | **1** | **1** |
| 16=4 x 4 | 256 | 8.528 | 2.520 | **3.38** | 2 | **1.260** | **2.520** |
| 64=8 x 8 | 256 | 14.641 | 5.852 | **2.50** | 5 | **1.170** | **5.648** |
| 256=16 x 16 | 256 | 28.848 | 12.070 | **2.39** | 10 | **1.207** | **11.574** |
| 1024=32 x 32 | 256 | 58.438 | 23.367 | **2.50** | 21 | **1.113** | **22.648** |
| 4096=64 x 64 | 256 | 128.087 | 45.199 | **2.83** | 42 | **1.076** | **44.082** |
| 16384=128 x 128 | 128 | 300.330 | 87.789 | **3.42** | 85 | **1.033** | **86.668** |

### 5.2 Efficiency of Adaptive Routing

The adaptive routing protocol was designed in order to speed up the message set transfer time and to avoid deadlocks during the run, although this could not

be proved. When an agent computes a new direction and whenever the channel in that direction is occupied, the agent chooses the other direction if there is a choice at all.

**First Test Case.** For this scenario with a common target the performance of adaptive routing is the same as for the non-adaptive routing. The reason is that all routes to the target are heavily congested. This means that the adaptive routing needs not to be better, but it is also not worse for the investigated case.

**Second Test Case.** For this scenario with randomly chosen sources and targets, the adaptive routing performs slightly better. For example, for $N = 1024$, the message transfer time is reduced by 4.1%. There seems to be more potential to optimize the behavior of the agents (using an FSM, or using a larger neighborhood) in order to guide them in a way that six agents are almost constantly consumed by the target.

### 5.3 Deadlocks

A trivial deadlock can be produced if all $6N$ channels contain agents (fully packed), thus no moving is possible at all. Another deadlock appears if $M$ agents line up in a loop on all the channels belonging to one lane, and all of them have the same lane direction. Then the lane is completely full and the agents are stuck. To escape from such a deadlock would only be possible if the agents can deviate from the shortest path, e.g. by choosing a random direction from time to time. More interesting are the cyclic deadlocks where the agents form a loop and are blocking each other (no receiving channel is free in the loop). Two situations were investigated.

**First situation** `[Right Loop]`**.** An empty node $\Omega$ is in the center of six surrounding nodes, let us call them $A_0$, $A_1$, $A_2$, $A_3$, $A_4$, $A_5$ clockwise. Agent at $A_0$ wants to go to $A_2$, $A_1$ to $A_3$, $A_2$ to $A_4$, $A_3$ to $A_5$, $A_4$ to $A_0$, $A_5$ to $A_1$, in short the $A_i$ want to go to $A_{i+2}$ all around. Note that each agent has two alternatives: going first via $\Omega$ through the center or going first to a surrounding node (e.g., agent at $A_0$ can go first to $\Omega$ and then to $A_2$, or first to $A_1$ and then to $A_2$). Whether a deadlock appears depends on the initial assignments to the channels. If the initial assignments of all agents are "`use the left channel first`" via surrounding nodes, then the agents block each other cyclically. Otherwise they can move via the center node $\Omega$ and no deadlock occurs. Thereby it is assumed that the channels in $\Omega$ are empty or become empty after some time and are not part of other deadlocks.

**Second situation** `[Left Loop]`
This situation is symmetric to the right loop, except that the loop direction is now counter-clockwise. A deadlock will appear if the initial directions of all agents are "`use the right channel first`".

If an initial configuration includes a right loop and a left loop, then at least one deadlock will appear if the initial assigned channel is fixed to the left or to the right. There are several ways to dissolve such deadlocks:

1. One way is to randomize the initial channel assignment. This works usually but there exists a very low probability that all chosen initial directions still can produce this deadlock.
2. A spatial inhomogeneity is used, e.g., agents at "odd" nodes use initially the left subroute channel and agents at "even" nodes use the right subroute channel. The partition "odd–even" means $x+y \equiv 1$ or $x+y \equiv 0$ respectively, under addition modulo 2. This kind of partition, among others, was examined and did work for a limited set of experiments.
3. It would be possible to redistribute the channels during the run, by using a two-stage interaction-advection transition similar to the "FHP" Lattice Gas CA [16]: move / don't move, then redistribute. In this case, the initially assigned channels and the used channels during the run could be dynamically rearranged.

## 6 Conclusion

The properties of a family of scalable 6–valent triangular tori were studied herein and for this family a minimal routing protocol was defined. A novel router with six channels per node was modeled as cellular automata. Each agent has a computed direction defining the new channel in the adjacent node. The computed direction is a "minimal" direction leading on the shortest path to the target. The novel router is significantly faster (2.5 times for 1024 nodes) than an optimized reference router with one agent per node. In addition, an adaptive routing protocol was defined, which prefers the leftmost channel of a minimal route if the rightmost channel is occupied. Thereby a speed-up of 4.1% for 1024 nodes was reached. Further work can be done in order to optimize the adaptiveness, e.g. by using a genetic algorithm. Deadlocks may appear for special situations when the system is completely full, or full along one axis, or when agents form a loop. In order to avoid an initial deadlock, the initial setting of the channels were alternated in space. Thereby the constructed initial deadlock situation could be dissolved. In order to dissolve deadlocks securely, a random or pseudo-random component should be introduced that also may allow the agents to bypass congested routes. Further investigations are also planned to simplify the routing protocol by exploring the symmetries of the isotropic triangular grid: it is conjectured that this approach may drastically simplify the router, at least in a deterministic or adaptive context [17,18].

**Acknowledgement.** We are indebted to Patrick Ediger for supplying reference data on the performance of the $T$-grid with one cell per node and for valuable discussions with him.

## References


1. Ediger, P., Hoffmann, R., Désérable, D.: Routing in the Triangular Grid with Evolved Agents. In: High Perf. Comp. Sim (HPCS), Special Session on Cellular Automata, Algorithms and Architectures (CAAA), Caen, France, pp. 582–590 (2010)
2. Loh, P.K.K., Prakash, E.C.: Performance Simulations of Moving Target Search Algorithms. Int. J. Comput. Games Technol., 1–6 (2009)
3. Korf, R.E.: Real-Time Heuristic Search. Artif. Intell. 42(2-3), 189–211 (1990)
4. Ediger, P., Hoffmann, R.: Routing Based on Evolved Agents. In: 23rd PARS Workshop on Parallel Systems and Algorithms, Hannover, Germany, pp. 45–53 (2010)
5. Ediger, P., Hoffmann, R.: CA Models for Target Searching Agents. In: Proc. Automata 2009: 15th International Workshop on Cellular Automata and Discrete Complex Systems, São José dos Campos, Brazil, pp. 41–54 (2009)
6. Ediger, P., Hoffmann, R., Désérable, D.: Rectangular vs Triangular Routing with Evolved Agents. In: High Perf. Comp. Sim (HPCS), Special Session on Cellular Automata Algorithms and Architectures (CAAA), Istanbul, Turkey, pp. x1–x7 (2011)
7. Chen, M.-S., Shin, K.G., Kandlur, D.D.: Addressing, Routing and Broadcasting in Hexagonal Mesh Multiprocessors. IEEE Trans. Comp. 39(1), 10–18 (1990)
8. Désérable, D.: A Family of Cayley Graphs on the Hexavalent Grid. Discrete Applied Math. 93(2-3), 169–189 (1999)
9. Désérable, D.: Minimal Routing in the Triangular Grid and in a Family of Related Tori. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 218–225. Springer, Heidelberg (1997)
10. Désérable, D.: Broadcasting in the Arrowhead Torus. Computers and Artificial Intelligence 16(6), 545–559 (1997)
11. Heydemann, M.-C., Marlin, N., Pérennes, S.: Complete Rotations in Cayley Graphs. European Journal of Combinatorics 22(2), 179–196 (2001)
12. Désérable, D.: Arrowhead and Diamond Diameters (unpublished)
13. Dally, W.J., Seitz, C.L.: The Torus Routing Chip. Dist. Comp. 1, 187–196 (1986)
14. Hoffmann, R.: The GCA-w Massively Parallel Model. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 194–206. Springer, Heidelberg (2009)
15. Poupet, V.: Translating Partioned Cellular Automata into Classical Type Cellular Automata. Journées Automates Cellulaires,130–140 (2008)
16. Frisch, U., Hasslacher, B., Pomeau, Y.: Lattice-Gas Automata for the Navier–Stokes Equation. Phys. Rev. Lett. 56(14), 1505–1508 (1986)
17. Désérable, D.: Hexagonal Bravais–Miller Routing of Shortest Path (unpublished)
18. Miller, W.H.: A Treatise on Crystallography. For J. & J.J. Deighton (1839)

# Domain Specific Language and Translator for Cellular Automata Models of Physico-Chemical Processes*

Konstantin Kalgin

Supercomputer Software Department,
Institute of Computational Mathematics and Mathematical Geophysics,
Russian Academy of Sciences
kalgin@ssd.sscc.ru

**Abstract.** A new domain specific language CACHE and its translator into C and Processing are presented. The domain is a set of cellular automata models of physico-chemical processes. The language and the translator are intended for using by researchers studying such processes. The translator allows to obtain both serial and parallel programs on C language. Multicores and clusters as target parallel architectures are supported. Additionally, one can easily visualize the process interactively, create a movie, and publish a Java-applet in the Internet using Processing.

## 1 Introduction

Nowadays, cellular automata (CA) are widely used in investigation of physico-chemical processes on micro and nano levels, for example, oscillatory chemical surface reactions [1,2], absorption, sublimation and diffusion of atoms in the epitaxial growth processes [3]. Among modern CA simulation tools [4,5] there is not a single one suitable for studying such processes. This is because most of CA models of physico-chemical processes operate in asynchronous mode, and on different stages of investigation both serial and parallel implementations of the model are required. Serial implementation is needed on first stages of a CA model development when a researcher performs many small-size computational experiments, and parallel implementation is required later, when the researcher performs experiments with large cellular arrays to compare results with natural tests.

Taking into account the diversity of computer architectures, we can see that the following versions of implementation may be required by researchers:

- serial implementation with ability to visualize the process interactively,
- parallel implementation on a multicore computer with shared memory,
- parallel implementation on a cluster,
- parallel implementation on GPUs.

* Supported by (1)Presidium of Russian Academy of Sciences, Basic Research Program N 14-6 (2011), (2) Siberian Branch of Russian Academy of Sciences, SBRAS Interdisciplinary Project 32 (2009), (3) Grant RFBR 11-01-00567a.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 166–174, 2011.

To satisfy the above requirements an attempt is made: in this paper a new domain specific language CACHE (Cellular Automata for CHEmical models) and its translator into C and Processing [6] is presented. The domain is a set of cellular automata models of physico-chemical processes. The language and the translator are intended for studying such processes. The translator allows to obtain both serial (C, Processing) and parallel (C+MPI, C+POSIX Threads) implementations of a cellular automata model. Additionally, one can easily visualize the process interactively, create movie, and publish Java-applet in the Internet using Processing.

## 2 A Cellular Automaton Definition

### 2.1 Formal Representation of CA

Cellular automaton is specified by the following tuple:

$$CA = \langle Z^d, A, \Theta, M \rangle \tag{1}$$

where $Z^d$ is a finite set of *cell coordinates*, $A$ is an *alphabet*, i.e. a finite set of cell states, $\Theta$ is a *local transition rule*, and $M$ is an *iteration mode*.

Further we use two dimensional rectangular space $Z^2$:

$$Z^2 = \{(i,j) \mid 1 \le i \le N_x,\ 1 \le j \le N_y\} \tag{2}$$

with periodic boundaries. A pair $(\mathbf{x}, a) \in Z^2 \times A$ is called a *cell*, where $a \in A$ is a cell state, $\mathbf{x} \in Z^2$ is its coordinates. Set of cells $\Omega = \{(\mathbf{x_i}, a_i)\} \subset Z^2 \times A$ is called a *cellular array* if there does not exist a pair of cells with equal coordinates and $\{\mathbf{x} \mid (\mathbf{x}, a) \in \Omega\} = Z^2$. Since between the cells in a cellular array and their coordinates there exists a one-to-one correspondence, we further identify each cell with its coordinates.

The *local transition rule* $\Theta$ is a probabilistic function:

$$\Theta : A^{|B_\Theta|} \times A^{|C_\Theta|} \xrightarrow{p} A^{|B_\Theta|}, \tag{3}$$

where the *base template* $B_\Theta$ and the *context template* $C_\Theta$ are lists of *naming functions* $\phi : Z^2 \to Z^2$, $B_\Theta = \{\phi_1^B, \phi_2^B, \ldots, \phi_{|B_\Theta|}^B\}$, $C_\Theta = \{\phi_1^C, \phi_2^C, \ldots, \phi_{|C_\Theta|}^C\}$. These templates determine *base* and *context neighborhoods* of a cell $\mathbf{x}$:

$$B_\Theta(\mathbf{x}) = \{\phi_1^B(\mathbf{x}), \ldots, \phi_{|B_\Theta|}^B(\mathbf{x})\}, \quad C_\Theta(\mathbf{x}) = \{\phi_1^C(\mathbf{x}), \ldots, \phi_{|C_\Theta|}^C(\mathbf{x})\}. \tag{4}$$

Additionally, context and base templates should meet the following condition:

$$\forall \mathbf{x} \in Z^2 \ :\ B_\Theta(\mathbf{x}) \cap C_\Theta(\mathbf{x}) = \emptyset. \tag{5}$$

Further the following neighborhoods are used:

$$T_{13}(\mathbf{x}) = \{\mathbf{x} + \mathbf{v_0}, \mathbf{x} + \mathbf{v_1}, \ldots, \mathbf{x} + \mathbf{v_{12}}\}, \tag{6}$$

$$T_9(\mathbf{x}) = \{\mathbf{x}+\mathbf{v_0}, \mathbf{x}+\mathbf{v_1}, \ldots, \mathbf{x}+\mathbf{v_8}\}, \tag{7}$$

$$T_5(\mathbf{x}) = \{\mathbf{x}+\mathbf{v_0}, \mathbf{x}+\mathbf{v_1}, \ldots, \mathbf{x}+\mathbf{v_4}\}, \tag{8}$$

$$T_1(\mathbf{x}) = \{\mathbf{x}+\mathbf{v_0}\}, \tag{9}$$

where $V = \{\mathbf{v_0}, \mathbf{v_1}, \ldots, \mathbf{v_{12}}\} = \{(0,0), (0,1), (1,0), (0,-1), (-1,0), (1,1), (1,-1), (-1,1), (-1,-1), (0,2), (2,0), (0,-2), (-2,0)\}$ (Fig. 1).

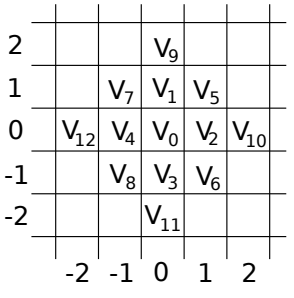



**Fig. 1.** Neighborhood $T_{13}$([6])

An *application of the local transition rule* ([3]) *to a cell* $\mathbf{x}$ results in replacing states of cells from the base neighborhood $B_\Theta(\mathbf{x})$ with next states $\Theta(S_\Theta)$, $S_\Theta(\mathbf{x})$ being a list of states of cells from $B_\Theta(\mathbf{x}) \cup C_\Theta(\mathbf{x})$.

Simulation process of CA is split into *iterations*. An iteration comprises $|Z^2| = N_x \cdot N_y$ local transition rule applications. The iteration mode $M$ defines the order of $\Theta$ application. There are many iteration modes. Here we use three of them: synchronous, asynchronous, and block-synchronous modes. In *synchronous* mode a local transition rule is applied simultaneously to all cells, this means that all cell states depend only on states from previous iteration. In *asynchronous* mode a local transition rule is sequentially applied $|Z^2|$ times to randomly chosen cells. To define *block-synchronous* mode we have to define a partition $\{S_0, S_2, \ldots, S_{w-1}\}$ of the set of cell coordinates $Z^2$ with additional condition to be met by each subset $S_k$:

$$\forall \mathbf{x}_1, \mathbf{x}_2 \in S_k \,:\, B_\Theta(\mathbf{x}_1) \cap C_\Theta(\mathbf{x}_2) = \emptyset. \tag{10}$$

In block-synchronous mode each iteration consists of $w$ stages. At each stage a local transition rule is simultaneously applied to all cells from randomly chosen subset $S_k$.

### 2.2 Formal Representation of Local Transition Rule

Usually, local transition rules of CA models of physico-chemical processes are expressed as a *composition of elementary substitutions* [7], where each elementary substitution corresponds to elementary physico-chemical process (diffusion,

adsorption, desorption, reaction, etc.). The composition of elementary substitutions is called a *composed substitution*. Before formal definition of composed substitution let us define the *elementary substitution*.

*Elementary substitution* $\Theta_{sub}$ is a probabilistic function $\Theta_{sub} : A^{|B_{\Theta_{sub}}|} \times A^{|C_{\Theta_{sub}}|} \xrightarrow{p} A^{|B_{\Theta_{sub}}|}$ with its own base and context templates, which is written in the following form:

$$\Theta_{sub} : \{a_1, a_2, \ldots, a_m\} \xrightarrow[cond]{p} \{a'_1, a'_2, \ldots, a'_{m'}\}, \tag{11}$$

where $m = |B_{\Theta_{sub}}| + |C_{\Theta_{sub}}|$, $m' = |B_{\Theta_{sub}}|$, $a_1, \ldots, a_m$ are current states of cells from base and context neighborhood, and $a'_1, \ldots, a'_{m'}$ are next states of cells from the base neighborhood. Application of $\Theta_{sub}$ to a cell $\mathbf{x}$ results in changing the states of cells $B_{\Theta_{sub}}(\mathbf{x})$ with probability $p$ only if condition $cond(a_1, a_2, \ldots, a_m)$ satisfied. Here probability $p$ and states $a'_i$ may be functions of current states, $p = p(a_1, a_2, \ldots, a_m)$, $a'_i = f_i(a_1, a_2, \ldots, a_m)$.

Let us return to the definition of a *composed substitution*. A composed substitution is a set of several elementary and/or other composed substitutions combined by composition rules. The most used rules of composition are *random execution* ($R$), *sequential execution* ($S$), and *randomly ordered sequential execution* ($RS$). These rules are given by

$$\Theta_R = R(\Theta_1, p_1; \Theta_2, p_2; \ldots, \Theta_n, p_n), \tag{12}$$

$$\Theta'_R = R(\Theta_1, \Theta_2, \ldots \Theta_n), \tag{13}$$

$$\Theta_S = S(\Theta_1, \Theta_2, \ldots \Theta_n), \tag{14}$$

$$\Theta_{RS} = RS(\Theta_1, \Theta_2, \ldots \Theta_n). \tag{15}$$

Additionally, for each $\Theta_i$, $i = 1 \ldots n$, a cell to which $\Theta_i$ is applied should be defined by a function $\varphi_i : Z^2 \to Z^2$:

$$B_{\Theta_R}(\mathbf{x}) = B_{\Theta'_R}(\mathbf{x}) = B_{\Theta_S}(\mathbf{x}) = B_{\Theta_{RS}}(\mathbf{x}) = \bigcup_{i=1}^{n} B_{\Theta_i}(\varphi_i(\mathbf{x})), \tag{16}$$

$$C_{\Theta_R}(\mathbf{x}) = C_{\Theta'_R}(\mathbf{x}) = C_{\Theta_S}(\mathbf{x}) = C_{\Theta_{RS}}(\mathbf{x}) = \bigcup_{i=1}^{n} C_{\Theta_i}(\varphi_i(\mathbf{x})) \setminus \bigcup_{i=1}^{n} B_{\Theta_i}(\varphi_i(\mathbf{x})). \tag{17}$$

The result of $\Theta_R$ application to $\mathbf{x}$ coincides with result of $\Theta_i$ application to $\varphi_i(\mathbf{x})$ with probability $\frac{p_i}{\sum_{k=1}^{n} p_k}$. If $p_i$ are all equal then they may be omitted (13). The result of $\Theta_S$ application to $\mathbf{x}$ coincides with sequential applications of $\Theta_1, \Theta_2, \ldots, \Theta_n$ to $\varphi_1(\mathbf{x}), \varphi_2(\mathbf{x}), \ldots, \varphi_n(\mathbf{x})$, respectively. The result of $\Theta_{RS}$ application to $\mathbf{x}$ coincides with sequential applications of randomly ordered $\Theta_{\tau_1}, \Theta_{\tau_2}, \ldots, \Theta_{\tau_n}$ to $\varphi_{\tau_1}(\mathbf{x}), \varphi_{\tau_2}(\mathbf{x}), \ldots, \varphi_{\tau_n}(\mathbf{x})$, , respectively.

Sometimes it is useful to define templates of a substitution $\Theta'$ on a group of cells, rather than on a single one:

$$T_{\Theta'}(\mathbf{x}_1, \ldots, \mathbf{x}_m) = \{\phi'_1(\mathbf{x}_1, \ldots, \mathbf{x}_m), \ldots, \phi'_{|T_{\Theta'}|}(\mathbf{x}_1, \ldots, \mathbf{x}_m)\}, \tag{18}$$

where $\phi'_i : \underbrace{Z^2 \times \ldots \times Z^2}_{m} \to Z^2$. Definition of elementary and composed substitutions, and its applications can be easily modified according to this template form. Further the following templates are used:

$$T_2(\mathbf{x}_1, \mathbf{x}_2) = \{\mathbf{x}_1, \mathbf{x}_2\}, \tag{19}$$

$$T_4(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}, \tag{20}$$

$$T'_5(\mathbf{x}_1, \ldots, \mathbf{x}_5) = \{\mathbf{x}_1, \ldots, \mathbf{x}_5\}. \tag{21}$$

## 3 CACHE Language

In the CACHE language the following concepts are to be defined: iteration mode, type of cell state, several parameters, local transition rule, several counters, initialization function, and color determination function. Description syntax of initialization function and color determination function is not considered here.

### 3.1 Iteration Mode

Cellular automata iteration mode $M$ is defined by one line:

Ex.1: `iteration : asynchronous(K)`

Here `K` may be a natural number or an arithmetic expression which means that each iteration of the model comprises `K` iterations as defined in Section 2. The following modes now are supported: `asynchronous`, `synchronous`, `blocksynchronous5`, `blocksynchronous9`, `blocksynchronous13`, `blocksynchronous25`. Each mode `blocksynchronousW`, where `W` is one of `5`, `9`, or `13`, corresponds to the template $T_W$ (6-8). For the block-synchronous modes the sets $S_k$ (10) are defined as follows:

$$S^{blocksynchronous5}_{k=0\ldots4} = \{(i,j) : \ i + 3j = k \,(mod\ 5)\}, \tag{22}$$

$$S^{blocksynchronous9}_{k=0\ldots8} = \{(i,j) : \ i - \left\lfloor \frac{i}{3} \right\rfloor + 3(j - \left\lfloor \frac{j}{3} \right\rfloor) = k \,(mod\ 9)\}, \tag{23}$$

$$S^{blocksynchronous13}_{k=0\ldots12} = \{(i,j) : \ i + 5j = k \,(mod\ 13)\}, \tag{24}$$

$$S^{blocksynchronous25}_{k=0\ldots24} = \{(i,j) : \ i - \left\lfloor \frac{i}{5} \right\rfloor + 5(j - \left\lfloor \frac{j}{5} \right\rfloor) = k \,(mod\ 25)\}. \tag{25}$$

### 3.2 Type of Cell State

Cell state is of primitive, set, or structure type. A primitive type is one out of: `byte`, `int` or `float`. In the case of set type, a set of possible cell state values is

defined by a list of identifiers. The structure type is a list of fields, where each field has a name and a type. A field is also of primitive, set, or structure type. Several examples of cell state type definition:

Ex.2:
```
cell : int;
cell : set{ empty, stable, movable };
cell : struct{
  h : int;
  a : set{ CO, O, Oss, CO_Oss, Empty };
};
```

### 3.3 Parameters

Description of a parameter starts from `'param'`. Parameter is a variable of primitive type visible among all arithmetico-logical expressions in cellular automata definition. Value of parameter by default is to be written in the text as follows:

Ex.3:
```
param float probCOdes = 0.2;
param float probO2ads = 0.1;
```

In addition to user-defined parameters, there are two built-in integer parameters `N` and `M`: sizes of cellular array $N_y$ and $N_x$, respectively. Default value for `N` and `M` is `100`.

The default value of a parameter can be changed by program arguments:

Ex.4: `./ca -probCOdes 0.5 -N 1024 -M 128`

### 3.4 Elementary Substitution

In the language base and context templates of elementary substitution (11) are

$$B_{\Theta_{sub}}(\mathbf{x}_1, \ldots, \mathbf{x}_m) = \{\mathbf{x}_1, \ldots, \mathbf{x}_{m'}\}, \tag{26}$$

$$C_{\Theta_{sub}}(\mathbf{x}_1, \ldots, \mathbf{x}_m) = \{\mathbf{x}_{m'+1}, \ldots, \mathbf{x}_m\}. \tag{27}$$

Template of a composed substitution is automatically computed according to (16) and it is not given directly in the text of cellular automata definition.

Description of an elementary substitution starts from `'sub'`. Two lists including initial and next cell states enclosed in square brackets are separated by `'->'`. Structured state is written as a list of field values enclosed in round brackets. In the case of states of structure/primitive type each field/state is stated as a concrete value, or as a designator, or as any-value expression. A concrete value is defined by arithmetic expression. A designator is an identifier not defined as parameter or as element of a set type. The any-value expression is written as a dot. The any-value expression in the list of initial cell states means "this value is not important", and in the list of next cell states means "do not change this value". Probability of actual state changing is represented by arithmetic expression which follows the symbol '`%`'. Condition of actual state changing of the elementary substitution is represented by an explicit and an implicit part. The explicit part of the condition is written in round brackets of `if` statement.

The implicit part is derived from designators and concrete values/states listed in the list of initial cell states.

Consider some examples of elementary substitution definition:

Ex.5:
```
sub stabilization: [ movable, a, b, c, d ]->[ stable ] %0.1
                   if( a==stable||b==stable||c==stable||d==stable )
```

Here `stabilization` is the name of the elementary substitution. Base template of the substitution is $T_1$ (9), and context template is $T_5' \backslash T_1$ according to (26,27). In the first square brackets, `movable` is a concrete value of the first cell state, `a`, `b`, `c` and `d` are designators of other four cell states. In the second square brackets next concrete state `stable` of the first cell is indicated. The last four cell states are not changed by application of the substitution. Implicit part of condition is represented by indicating the concrete value `movable` in the first square brackets: it means that state of the first cell is to be `movable`.

Ex.6:
```
sub ads02: [ (n>0 , Empty), (n, Empty) ] -> [ (., 0), (., 0) ]
```

Base template of the substitution is $T_2'$ (19), and context template is empty according to (26,27). Here two occurrences of `n` means that values of the first fields of two structured cell states is to be equal. It is the second way to specify implicit part of condition. The third way to specify implicit part of condition is using `A B C` expressions, where `A` is a designator of the value, `C` is an arithmetic expression, and `B` is one of `>`, `<`, `>=`, or `<=`.

### 3.5 Composed Substitution

Description of a composed substitution starts from '`csub`', which is followed by its name and a list of identifiers in round brackets. These identifiers are designators of cells to which the substitution are applied. To compose several substitutions one is to write the name of a composition rule (`random` for *random execution*, `seq` for *sequential execution*, and `random_order` for *randomly ordered sequential execution* rule) and then a list of rule names with arguments. A comma-separated list of rule names is enclosed in curly brackets. A comma-separated list of arguments is enclosed in round brackets. The arguments determine cells the substitution is to be applied to. Probabilities for substitution executions, which may be omitted in `random` composition rule, are represented by arithmetic expression after '`%`' symbol.

Ex.7:
```
csub rule( x ) : random{
          ads02(x, x+T5(1)) %0.25,
          ads02(x, x+T5(2)) %(0.5/2.0),
          ads02(x, x+T5(3)) %(1.0/4.0),
          ads02(x, x+T5(4)) %0.25
      }
```

Here an expression `x+T5(i)` determines i'th neighboring cell from $T_5$(`x`) (8). In the language neighborhood functions `TW(i)`, where `W` is on of `5`, `9`, `13`, `and 25`, are also implemented. For this example the base template of the composed substitution `rule` is $T_5$, and context template is empty according to (16) and (26,27).

### 3.6 Local Transition Rule

In the language the local transition rule is a composed substitution named `rule` with base and context templates on one cell (4).

### 3.7 Counters

Counter is a variable of integer type. Value of a counter equals to the number of cell states with certain properties in the cellular array. Values of all defined counters are printed at each iteration. Counters are very useful to analyse the modeling process in dynamics.

Description of counter starts from `'counter'` followed by its name, cell state and logical expression. The logical expression determines the explicit part of the condition of counter incrementing. The implicit part of the condition is represented in cell state description like it is represented in cell state description of elementary substitution (Sec. 3.4).

Ex.8:
```
counter howManyStableStates: [ stable ]
counter countAds0: [(h , C0)] if( h > 0 )
```

## 4 Translator

The translator allows to obtain both serial (C, Processing) and parallel (C+POSIX Threads for multicores, C+MPI for clusters) implementations of a cellular automata model. Parallel algorithm for asynchronous cellular automata simulation is described in [8]. Techniques for efficient parallel implementation of synchronous and block-synchronous cellular automata on multicore and cluster are considered in [9].

Target language (C, Processing) and computer architecture (serial, multicore, and cluster) are specified by command-line arguments:

Ex.9:
```
./cache -lang C -arch cluster -o file_mpi.c file.ca
./cache -lang Processing -o ca_model/ca_model.pde file.ca
```

A generated file consists of three parts:

- header (global variables definition and main function),
- implementation of local transition rule,
- implementation of iteration function.

The generated code for the first two parts depends quite slightly on target language and computer architecture. Differences are only in functions declaration, and cellular array declaration and allocation. Generation of the last part, which implements the iteration function, depends on iteration mode, target language and target computer architecture, not depending on the first two parts. All variants of iteration implementation (*iteration mode* $\times$ *target language* $\times$ *computer architecture*) are included in the translator. During the generation process the translator copies appropriate implementation to the output file.

## 5 Conclusion

The CACHE language and functionality of its translator are considered in some aspects. These tools allow to describe and to study CA models in helpful way on different parallel computer architectures. Now we have implemented and tested several physico-chemical models. The further work is to support GPUs as a target parallel architectures.

## References

1. Danielak, R., Perera, A., Moreau, M., Frankowicz, M., Kapral, R.: Surface Structure and Catalytic CO Oxidation Oscillations. arXiv:chao-dyn/9602015v1 (1996)
2. Elokhin, V.I., Latkin, E.I., Matveev, A.V., Gorodetskii, V.V.: Application of Statistical Lattice Models to the Analysis of Oscillatory and Autowave Processes on the Reaction of Carbon Monoxide Oxidation over Platinum and Palladium Surfaces. Kinetics and Catalysis 44(5), 672–700 (2003)
3. Neizvestny, I.G., Shwartz, N.L., Yanovitskaya, Z.S., Zverev, A.V.: 3D-model of epitaxial growth on porous {111} and {100} Si surfaces. Computer Physics Communications 147, 272–275 (2002)
4. Talia, D., Naumov, L.: Parallel Cellular Programming for Emergent Computation. In: Hoekstra, A.G., et al. (eds.) Simulating Complex Systems by Cellular Automata, Understanding Complex Systems. Springer, Heidelberg (2010)
5. WinALT system, `http://winalt.sscc.ru`
6. Processing, `http://processing.org`
7. Achasova, S.M., Bandman, O.L., Markova, V.P., Piskunov, S.V.: Parallel Substitution Algorithm. Theory and Application. World Scientific, Singapore (1994)
8. Kalgin, K.V.: Comparative study of parallel algorithms for asynchronous cellular automata simulation on different computer architectures. In: Bandini, S., Manzoni, S., Umeo, H., Vizzari, G. (eds.) ACRI 2010. LNCS, vol. 6350, pp. 399–408. Springer, Heidelberg (2010)
9. Kalgin, K.V.: Implementation of algorithms with a fine-grained parallelism on GPUs. Numerical Analysis and Applications (2011), doi:10.1134/S1995423911010058

# Dynamic Load Balancing for Lattice Gas Simulations on a Cluster*

Yuri Medvedev

The Institute of Computational Mathematics and Mathematical Geophysics,
Supercomputer Software Dept., pr. Acad. Lavrentiev, 6, 630090, Novosibirsk, Russia
medvedev@ssd.sscc.ru

**Abstract.** A problem of dynamic load balancing application for simulation of gas and fluid flows by lattice gas automata (LGA) is considered. The choice of a diffusion balancing method is justified. Results of testing both balanced and imbalanced cases are presented. Efficiency of the realizations for simulation the LGA and the PIC-method is compared.

**Keywords:** Cellular automata, LGA, parallel program, dynamic balancing.

## 1 Introduction

Lattice Gas Automata (LGA) are an effective model of simulation of fluids and gases flows. They are easily computed on a cluster because cells perform identical discrete functions. Traditional LGA models [1] have the Boolean alphabet. For them transition table can be easily located in the RAM, and cell transition is a simple substitution of a new value instead of an old one. So, processing time of each cell does not depend on its state. For the multi-particle LGA models with the integer alphabet proposed in [2], the amount of transition rules of a cell is considerably enlarged. The transition table can not be located in the RAM. Therefore, each transition has to be computed ad-hoc. Moreover, computation time becomes dependent essentially on an amount of particles in a cell. So, high particle density areas are significantly more time-consuming for computation, than the low density ones. It leads to load imbalance. So, cells are to be distributed proportionally to transition function computing time to make equal core loadings. But at the next iteration, particles move to adjacent cores, and there occur the imbalance again. For blast wave simulation this problem is particularly important [3]. In this case, the only way to solve the problem is to use a dynamic load balancing.

In this paper, a dynamic load balancing algorithm is proposed. Results of computer experiments are described. Efficiency of parallel realization with the balancing is obtained and compared with the efficiency without balancing. Also, balancing results of the LGA implementation is compared with that of PIC-method.

* Supported by 1) Presidium of RAS, Basic Research Program No 14–6 (2011), 2) Siberian Branch of RAS, Interdisciplinary Project No 32 (2011), 3) Grant RFBR 11–01–00567a.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 175–181, 2011.

## 2 Describing of the LGA Models

A 2D cellular array has a size $I \times J$. Lines are indexed by $i$ on the interval $0 \le i < I$. Columns are indexed by $j$ on the interval $0 \le j < J$. Each cell has 6 neighbors distanced by 1, i.e. the LGA has hexagonal pattern. A cell state is represented by a 6-th digits long vector. Each digit of the state vector is associated with one of neighboring cells. In traditional LGA [1] Boolean state vectors are used. Each component of this vector stores information about presence of a discrete model particle with the unit mass and the unit velocity. A velocity is directed to the neighboring cell corresponding to a component of the state vector. Multi-particle LGA [2] admit many particles with equally directed unit velocity to be located in a single cell. So, their state is represented by a vector with integer components.

The cellular automaton runs in the synchronous mode. It has two-staged iteration:

1. *Collisions*. Cells are processed independently. In each cell particles are "intermixed" in such a way that their total mass and total momentum are conserved. If the acceptable state does not exist, then it is an identical collision in which the resulting state is equal to the current one. If there is only one nonidentical state saving mass and momentum, the cell transition to the state is deterministic. If there exist $N > 1$ such states, the cell transits to one of them with probability $1/N$.

2. *Propagation*. The result of this stage depends on the states of neighboring cells. Each particle propagates to a neighbor according to its velocity vector direction.

After appropriate number of iterations, averaging of particles velocity and concentration is performed for obtaining the flow field.

### 2.1 Features of the Multi-particle Models

Not in all tasks traditional LGA [1] with the Boolean alphabet yield good effect of simulating. So, it is impossible to use them for simulating processes with the heavy gradients of pressure (for example, detonation). Also, they cannot simulate flows with moving obstacles. These limitations are absent in the multi-particle model [2]. Since its state is represented by an integer vector, a cell has a huge number of possible various states. It is impossible to find all the states conserving the mass and the momentum of particles in a cell at each collision execution. Therefore one has to compute a new state ad-hoc. From the above, it follows that the multi-particle LGA has the following properties:

1. The collisions take up more time, than in Boolean case;
2. Runtime of the collision functions depends on a cell state essentially;
3. If the program is run on a cluster, there may be a load imbalance.

### 2.2 Parallel Program Implementation

In the parallel implementation of the LGA the domain decomposition method is used. The cellular array is divided into stripes along the axis $i$. It is enough for good performance to use one dimension cuts in this case. To any of $p$ cores ($0 \le rank < p$) a stripe is assigned for processing, which consists of several lines $i_A(rank) \le i < i_B(rank)$. Boundaries of the stripes are related by the following ratios:

$$i_A(rank + 1) = i_B(rank), \text{ for } 0 \le rank < p - 1,$$

$$i_A(0) = 0,\ i_B(p) = I.$$

In the absence of balancing the stripes have equal width (exactly within integer rounding, $\lfloor x \rfloor$ denotes floor $x$):

$$i_A(rank) = \lfloor rank \cdot I / p \rfloor, \text{ for } 0 \le rank < p - 1,$$

$$i_B(rank) = \lfloor (rank + 1) \cdot I / p \rfloor, \text{ for } 1 \le rank < p.$$

Iteration consists of the following three stages:

1. Collisions.
2. Exchanging of boundary cells states.
   The core with the number *rank* sends:
   to the core $rank - 1$ the line $i_A(rank)$, for $1 \le rank < p$,
   to the core $rank + 1$ the line $i_B(rank) - 1$, for $0 \le rank < p - 1$.
   The core with the number *rank* receives:
   from the core $rank - 1$ the line $i_A(rank) - 1$, for $1 \le rank < p$,
   from the core $rank + 1$ the line $i_B(rank)$, for $0 \le rank < p - 1$.
   The received values are used at the third stage.
3. Propagation. For boundary cells in a stripe, the set of its neighbors includes those ones, whose states are just received from adjacent cores.

## 3 Load Balancing Algorithm

From the variety of balancing algorithms [4] we have selected the diffusion algorithm for the following reasons.

1. The algorithm has high efficiency of parallel implementation on a cluster with the size of thousands cores, as distinct to the centralized algorithms, where such size brings significant overheads.
2. The place of a section of the cellular array does not influence the amount of the sent data at each iteration.
3. Adjacent cores, as well, exchange their boundaries at the each iteration.
4. The load imbalance increases slowly (except in the tasks with explosive wave).

### 3.1 Initial Balancing

Before the simulator starts, one test iteration is performed for every line $i$ ($0 \le i < I$). At this iteration, runtime of the collision function in the lines is computed as follows: $t_0(i) = \sum_{j=0}^{J-1} t(i, j)$, where $t(i,j)$ is the runtime of the collision function in a cell $(i,j)$. The propagation stage requires less time than the collision one and is not taken into account, hence, the averaged core time is computed as follows: $\bar{t} = \sum_{i=0}^{I-1} t_0(i) \Big/ p$. Now for all cores ($0 \le rank < p$) stripe boundaries $i_A(rank)$ and $i_B(rank)$ may be calculated. The lower boundary $i_A$ is selected so that the following conditions are satisfied:

$$\sum_{i=0}^{i_A(rank)} t_0(i) \le rank \cdot \overline{t}\ , \quad \sum_{i=0}^{i_A(rank)+1} t_0(i) > rank \cdot \overline{t}\ .$$

The choice of the upper boundary $i_B$ should be made as follows:

$$\sum_{i=0}^{i_B(rank+1)} t_0(i) \le rank \cdot \overline{t}\ , \quad \sum_{i=0}^{i_B(rank+1)+1} t_0(i) > rank \cdot \overline{t}\ .$$

That is, the cells located between the lower and the upper boundaries, are processed in the same time $\overline{t}$. Let us remind that as before $i_A(0) = 0$, $i_B(p) = I$. This method allows setting precisely enough the initial balancing in conditions when dependence of a cell processing time on its state is unknown.

### 3.2 Dynamic Balancing

The three stages of iteration are described in section 2.2. The first and third stages with balancing remain without modifications. The second stage has three small differences relative to the case without balancing.

1. The boundaries are doubled in thickness. It leads to a little less than twofold magnification of the transfer time since the transfer is carried out by one portion. The core with the number *rank* sends:

   to the core $rank - 1$ the lines $i_A(rank)$ and $i_A(rank) + 1$, for $1 \le rank < p$,
   to the core $rank + 1$ the lines $i_B(rank) - 2$ and $i_B(rank) - 1$, for $0 \le rank < p - 1$.

   The core with the number *rank* receives:

   from the core $rank - 1$ the lines $i_A(rank) - 2$ and $i_A(rank) - 1$, for $1 \le rank < p$,
   from the core $rank + 1$ the lines $i_B(rank)$ and $i_B(rank) + 1$, for $0 \le rank < p - 1$.

2. Together with the states of boundary cells, the runtime of the just completed collisions stage $t^{(k)}(rank)$ is transmitted, where $k$ is the iteration number.
3. After the transfer, an adjustment of the boundaries of the stripes in the each core is performed.

$$i_A^{(k+1)}(rank) = i_A^{(k)}(rank) - 1, \text{ if } t^{(k)}(rank) / t^{(k)}(rank - 1) < b, \text{ for } 1 \le rank < p,$$
$$i_A^{(k+1)}(rank) = i_A^{(k)}(rank) + 1, \text{ if } t^{(k)}(rank - 1) / t^{(k)}(rank) < b, \text{ for } 1 \le rank < p,$$
$$i_B^{(k+1)}(rank) = i_B^{(k)}(rank) - 1, \text{ if } t^{(k)}(rank + 1) / t^{(k)}(rank) < b, \text{ for } 0 \le rank < p - 1,$$
$$i_B^{(k+1)}(rank) = i_B^{(k)}(rank) + 1, \text{ if } t^{(k)}(rank) / t^{(k)}(rank + 1) < b, \text{ for } 0 \le rank < p - 1,$$

where $b$ is a movement borders threshold ($0 \le b \le 1$). The argument $b$ defines the value of imbalance of the time between adjacent cores, above which the boundary drifts are performed in the direction of the less loaded core. At $b = 0$, we have a static balance case. At $b = 1$, sensitivity of the response is the highest.

It should be noted that, in our implementation, the overhead for the dynamic diffusion balancing is minimal for two following reasons.

1. Data transferring is carried out only to adjacent cores. Hence, the increase of cores number does not leads to efficiency saturation as it takes place in the centralized balancing.

2. There are no additional transfers in comparison to the unbalanced case. Only the size of a single package of transmitted data increases. This does not lead to a substantial increase of transfer time.

## 4 Results of Testing[1]

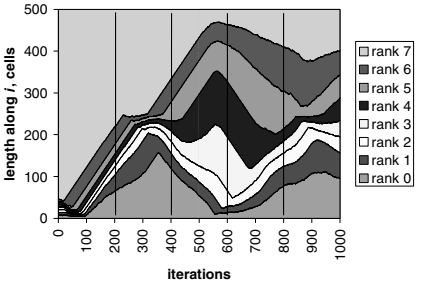



**Fig. 1.** Dynamics of the cores load

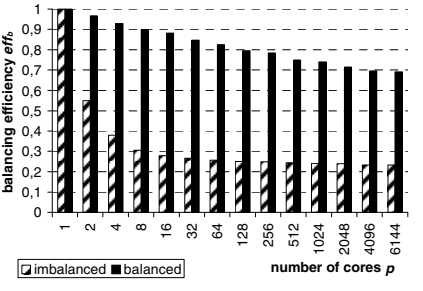



**Fig. 2.** Efficiency of dynamic balancing

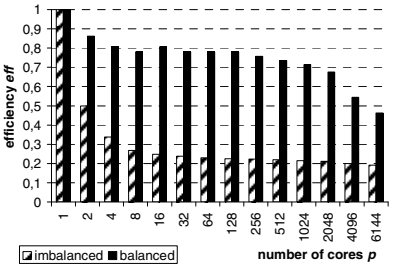



**Fig. 3.** Efficiency of parallel realization

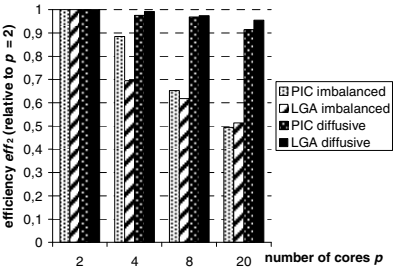



**Fig. 4.** Comparison with the PIC method

### 4.1 An Example of Explosion Simulation. Dynamics of Distribution of the Cellular Array among Cores

An explosion in a closed camera is simulated. The cellular array has the size $I = 500$, $J = 200$. The camera walls are allocated on the cellular array margins. Also obstacles are present in the middle of the array: [(250, 0), (250, 66)] and [(250, 133), (250, 199)]. Density in the cellular array at $i \le 50$ is equal to 82 particles per cell. Density in the rest of the array (at i>50) is equal to 14 particles per cell. The number of cores is $p = 8$. 1000 iterations have been completed. Initial balancing is fulfilled according to the method in the section 3.1. At each of the 8 cores, the boundaries of the stripes $i_A(rank)$ and $i_B(rank)$ have been updated at each iteration.

The process goes as follows. In the beginning, the dense gas moves along the cellular array to its middle part and hit against the obstacles. A part of the flow transits through the hole between the obstacles. Another part is reflected from the

[1] Tests have been performed on MVS100K — JSCC Cluster, Moscow.

obstacles. After that, the reflected flow goes backward, is reflected from the bottom boundary of the cellular array again, and goes up to the obstacles. The part of flow which has transited through the hole is shaped in two vortexes above the obstacles. The amount of the particles changes along $i$ quite intensively. Therefore, it is difficult to achieve acceptable efficiency of the balancing.

The chart of stripes width allocated on each core is given in Fig. 1. The efficiency also does not remain steady; therefore, for its estimate the next test with quieter flow has been performed.

### 4.2 An Example of a Quiet Flow. Comparison of the Efficiency

In the second test the program has been launched on $p = 1, 2, 4...$ 4096 and 6144 cores with two conditions: with balancing and without one. Sizes of the cellular array are equal to $I = 100 \cdot p$, $J = 10$. The size $I$ is taken depending on the number of the cores $p$ in order to eliminate harmful influence of the hardware over the runtime. The size $J$ is selected tiny to diminish the test execution time. At the initial state, the density of the particles is invariable along $j$ and is arranged linearly lengthways $i$ from the 70 particles per a cell (at $i = 0$) to the 7 particles per cell (at $i = I - 1$). These boundary conditions are provided throughout the whole simulation process. Hence, the flow escapes along $i$ in the positive direction. The special boundary cells $[(0, 0), (0, 9)]$ and $[(I - 1, 0), (I - 1, 199)]$ provide the demanded density of the particles. The walls $[(0, 0), (I - 1, 0)]$ and $[(0, 9), (I - 1, 9)]$ flanking the cellular array keep the flow inside.

Fig. 2 shows the efficiency of the balancing (averaged over 100 iterations):

$$eff_b = \sum_{rank=0}^{p-1} t(rank) \Big/ p \cdot \max\left\{t(rank) : rank = \overline{0, p-1}\right\},$$

where $t(rank)$ is the total runtime of the collision functions at the cells with $i_A(rank) \le i < i_B(rank)$, $(0 \le rank < p - 1)$. This efficiency indicates only the imbalance of the cores loading, not including the overhead of communication.

Fig. 3 shows the parallelization efficiency of the program with the overhead of communication (also averaged over 100 iterations): $eff = t_1 / t_p$, where $t_1$ is the executive time of the program on one core, $t_p$ is the executive time of the program on $p$ cores. Due to the fact that the size $I$ of the cellular array is proportional to the number of cores $p$, the factor $p$ in the denominator is missing.

Analysis of Fig. 2 and Fig. 3 leads to the following conclusions:

1. Without balancing the execution time, as expected, is greater than with balancing.
2. Efficiency remains at acceptable levels even for a large number of cores.
3. With balancing the time loss dependence on the number of cores is logarithmic.
4. Communications between the cores bring additional overhead of execution time, which increases with the number of cores.
5. On a great number of cores ($p > 2000$), the decrease of performance due to communications predominates the decrease generated by the imbalance of loading. This shows the applicability of the diffusion balancing algorithm on any number of cores. With the increase of cores number, the share of losses due to the imbalance is reduced and the share of losses in the communication increases.

### 4.3 Comparison of the Implementation Efficiency for the LGA and the PIC-Method

In [5], the diffusion algorithm of balancing for the method of particles in cells (PIC) is considered. There, in particular, program runtime on different number of cores with balancing and without it is given. The amount of particles used there, amounted to 800 thousand. We have done the test in which it was used 800 thousand cells, and have compared the efficiencies. Unfortunately, in [5] there is no data on runtime on one core, therefore efficiency was computed in relation to runtime on two cores: $eff_2 = 2 \cdot t_2 / p \cdot t_p$ , where $t_2$ is the program runtime on two cores, and $t_p$ is the program runtime on $p$ cores.

Fig. 4 shows, that efficiency of the diffusion algorithm of balancing used in the LGA is comparable with that used in the PIC-method.

## 5 Conclusion and Future Work

Our investigations show that when simulating flow using multi-particle LGA, dynamic load balancing is not only necessary but also can be successfully used. With the acceptable efficiency, the number of cores used is limited by the speed of communication rather than the load imbalance. In comparison with the balancing of the PIC method, balancing of the LGA showed reasonable results. It is worth noting, that in the future LGA models, the transitions rules will be more complicated, will more imbalanced, and balancing will more urgent.

## References

1. Bandman, O.L.: Cellular-Automata Models of Spatial Dynamics. System Informatics (10), 57–113 (2005) (in Russian)
2. Medvedev, Yu.: The FHP-MP model as multiparticle Lattice-Gas. Bulletin of the Novosibirsk Computing Center. Series: Computer Science (27), 83–91 (2008)
3. Medvedev, Yu.: Cellular-automaton simulation of a cumulative jet formation. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 249–256. Springer, Heidelberg (2009)
4. Hu, Y.F., Blake, R.J.: Load Balancing for Unstructured Mesh Applications (2000), `http://www.research.att.com/~yifanhu/PROJECT/pdcp_siam/pdcp_siam.html`
5. Kraeva, M.A., Malyshkin, V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. International Journal on Future Generation Computer Systems, devoted to Parallel Computing Technologies 17(6), 755–765 (2001)

# Efficient Associative Algorithm for Finding the Second Simple Shortest Paths in a Digraph

Anna Nepomniaschaya

Institute of Computational Mathematics and Mathematical Geophysics,
Siberian Division of Russian Academy of Sciences,
pr. Lavrentieva, 6, Novosibirsk, 630090, Russia
anep@ssd.sscc.ru

**Abstract.** We present an associative algorithm for finding the second simple shortest paths from the source vertex to other vertices of a directed weighted graph. Our model of computation (the STAR–machine) simulates the run of associative (content addressable) parallel systems of the SIMD type with vertical processing. We first propose the data structure that uses the graph representation as a list of triples (edge end–points and the weight) and the shortest paths tree obtained by means of the classical Dijkstra algorithm. The associative algorithm is given as the procedure SecondPaths, whose correctness is proved and the time complexity is evaluated.

**Keywords:** Directed weighted graphs, shortest paths tree, second simple shortest paths, associative parallel processor, access data by contents.

## 1 Introduction

Finding the shortest paths between two vertices is a well–studied graph problem. It can be efficiently solved using the classical Dijkstra algorithm [1] implemented by means of the Fibonacci heaps. The $k$ shortest paths problem is a natural generalization of the shortest paths problem when several shortest paths must be determined. Given a graph $G$ with $n$ vertices and $m$ edges, two vertices $s$ and $t$, and an integer $k$, one has to enumerate the $k$ shortest paths from $s$ to $t$ in the order of increasing their length.

For the problem of finding the $k$ simple (loopless) shortest paths, the fastest $O(k(m + n \log n))$ time algorithm for undirected graphs was proposed by Katoh et al. [5] and the best $O(kn(m + n \log n))$ time algorithms for directed graphs were proposed independently by Yen [12] and Lawler [6]. Such estimations are obtained using the modern data structures for implementing the Dijkstra algorithm. For unweighted directed graphs and for graphs with small integer weights, Roditty and Zwick [11] propose a randomized algorithm for finding the $k$ simple shortest paths that runs in $O(km\sqrt{n} \log n)$ time. The authors reduce the problem of finding the $k$ simple shortest paths to $O(k)$ computations of the second simple shortest path each time in a different subgraph of $G$. Gotthilf and Lewenstein [4] present an $O(k(mn + n^2 \log \log n))$ time algorithm for finding the

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 182–191, 2011.

$k$ simple shortest paths from $s$ to $t$ in weighted directed graphs using the efficient solution of the all–pairs shortest paths problem. In the case of directed graphs, Eppstein [2] proposes an efficient $O(m+n\log n+kn)$ time algorithm for finding the $k$ shortest paths (allowing cycles) from $s$ to each vertex of $G$. This algorithm builds an implicit representation of paths. The edges in any path can be explicitly listed in the time proportional to the number of edges.

In this paper, we propose an efficient associative algorithm for finding the second simple shortest paths from $s$ to all vertices of a directed weighted graph $G$. Our model of computation (the STAR–machine) simulates the run of associative (content addressable) parallel systems of the SIMD type with bit–serial (vertical) processing. Following Foster [3], we assume that each elementary operation of the STAR–machine (its microstep) takes one unit of time. We first propose a data structure and explain how to build it. Then we propose an associative algorithm for finding the second simple shortest paths from $s$ to all vertices of $G$. On the STAR–machine, this algorithm is implemented as procedure SecondPaths, whose correctness was proved. The procedure uses the graph representation as a list of triples (edge end-points and the weight) and the shortest paths tree as a bit–column that saves positions of the tree edges. The procedure SecondPaths returns a matrix $TPaths[2]$, whose every $i$-th column saves positions of edges belonging to the second simple shortest path from $s$ to $i$. We obtain that it takes $O(r(\log n + deg^+(G)))$ time, where $r$ is the number of non–tree edges that are really used for finding the second simple shortest paths and $deg^+(G)$ is the maximum number of edges outgoing from graph vertices.

## 2 Model of Associative Parallel Machine

In this section, we propose a brief description of our model which is based on a Staran–like associative parallel processor [3]. It is defined as an abstract STAR–machine of the SIMD type with vertical data processing [7]. The model consists of the following components:

- a sequential control unit (CU), where programs and scalar constants are stored;
- an associative processing unit consisting of $p$ single–bit processing elements (PEs);
- a matrix memory for the associative processing unit.

The CU passes an instruction to all PEs in one unit of time. All active PEs execute it while inactive PEs do not. Activation of a PE depends on the data.

Input binary data are loaded into the matrix memory in the form of two–dimensional tables, where each data item occupies an individual row and it is updated by a dedicated PE. We assume that the number of PEs is no less than the number of rows in any input table. The rows are numbered from top to bottom and the columns – from left to right. Both a row and a column can be easily accessed. Some tables may be loaded into the matrix memory.

An associative processing unit is represented as $h$ ($h \geq 4$) vertical registers, each consisting of $p$ bits. A vertical register can be regarded as a one–column

array. The STAR–machine runs as follows. The bit columns of the tabular data are stored in the registers which perform necessary Boolean operations.

To simulate data processing in the matrix memory, we use data types **word, slice**, and **table**. The types **slice** and **word** are used for the bit column access and the bit row access, respectively, and the type **table** is used for defining tabular data. Assume that any variable of the type **slice** consists of $p$ components which belong to $\{0, 1\}$. For simplicity, let us call *slice* any variable of the type **slice**.

Let us present the main operations for slices.

Let $X, Y$ be variables of the type **slice** and $i$ be a variable of the type **integer**. We use the following operations:

SET($Y$) simultaneously sets all components of $Y$ to $'1'$;

CLR($Y$) simultaneously sets all components of $Y$ to $'0'$;

$Y(i)$ selects the value of the $i$-th component of $Y$;

FND($Y$) returns the ordinal number $i$ of the first (the uppermost) bit $'1'$ of $Y$, $i \geq 0$;

STEP($Y$) returns the same result as FND($Y$) and then resets the first found $'1'$ to $'0'$.

To carry out the data parallelizm, we introduce in the usual way the bitwise Boolean operations: $X\ and\ Y$, $X\ or\ Y$, $not\ Y$, $X\ xor\ Y$. We also use the predicate SOME($Y$) that results in **true** if there is at least a single bit $'1'$ in the slice $Y$. For simplicity, the notation $Y \neq \emptyset$ denotes that the predicate SOME($Y$) results in **true**.

Note that the predicate SOME($Y$) and all operations for the type **slice** are also performed for the type **word**.

Let $T$ be a variable of the type **table**. We employ the following elementary operations:

ROW($i, T$) returns the $i$-th row of the matrix $T$;

COL($i, T$) returns its $i$-th column.

Note that the STAR statements are defined in the same manner as for Pascal. We will use them later for presenting our procedures.

Now, we recall two basic procedures [8] to be used later on. These procedures use the given global slice $X$ to indicate with the bit $'1'$ the row positions used in the corresponding procedure. In [8], we have shown that these procedures take $O(l)$ time each, where $l$ is the number of bit columns in the corresponding matrix.

The procedure MATCH($T, X, v, Z$) simultaneously defines positions of the given matrix $T$ rows which coincide with the given pattern $v$. It returns the slice $Z$, where $Z(i) =' 1'$ if and only if ROW($i, T$) $= v$ and $X(i) =' 1'$.

The procedure MIN($T, X, Z$) simultaneously defines positions of the given matrix $T$ rows, where the minimum entry is located. It returns the slice $Z$, where $Z(i) =' 1'$ if and only if ROW($i, T$) is the minimum matrix entry and $X(i) =' 1'$.

## 3 Preliminaries

Let $G = (V, E)$ denote a *digraph* with $n$ vertices and $m$ directed edges (arcs). We assume that $V = \{1, 2, \ldots, n\}$. Let $wt(e)$ denote a function that assigns a weight to every edge $e$. We assume that all arcs have a non-negative weight. Moreover, we do not allow self-loops and multiple edges.

An arc $e$ directed from $u$ to $v$ is denoted by $e = (u, v)$, where $u = tail(e)$ and $v = head(e)$. Let $deg^+(G)$ denote the maximum number of arcs outgoing from the graph vertices.

The *shortest path* from $v_1$ to $v_k$ is a finite sequence of vertices $v_1, v_2, \ldots, v_k$, where $(v_i, v_{i+1}) \in E$ $(1 \leq i < k)$, and the sum of weights of the corresponding arcs is minimum. Let $dist(v_1, v_k)$ denote the *length* of the shortest path from $v_1$ to $v_k$. If there is no path between these vertices, then $dist(v_1, v_k) = \infty$.

The *shortest paths tree* $T$ with a root $v_1$ is a connected acyclic subgraph of $G$ which includes all graph vertices, and for every vertex $v_j$ there is a unique shortest path from $v_1$. The arcs of $G$ that do not belong to $T$ are called *non–tree* edges. For any path $p$, let $sidetracks(p)$ be a sequence of non–tree edges that belong to $p$.

For every arc $(u, v)$ in $G$, Eppstein [2] defines a function $\delta(u, v) = wt(u, v) + dist(v_1, u) - dist(v_1, v)$. Informally, $\delta(u, v)$ shows how much distance is lost if, instead of taking the shortest path from $v_1$ to $v$, we first use the shortest path from $v_1$ to $u$ and then take the arc $(u, v)$. Clearly, for every $e \in G$, $\delta(e) \geq 0$, and for every $e \in T$, $\delta(e) = 0$. If $\delta(e)$ is considered as a weight function on the arcs of $G$, then the weight of every path $p$ will be equal to the sum of weights of the non-tree edges that appear in this path. Therefore, the problem of finding the $k$ shortest paths $p$ can be stated as problem of computing the $k$ smallest values of $\sum_{(u,v) \in sidetracks(p)} \delta(u, v)$.

## 4 Data Structure

In this section, we propose a data structure and explain how to obtain it. Observe that every second simple shortest path will include a single non–tree edge. Such a path will be obtained either by concatenating some shortest path with a non–tree edge or by concatenating a certain second simple shortest path with a suffix of some shortest path.

We will use the function $\delta(u, v)$, proposed by Eppstein [2], instead of the function $wt(u, v)$. On the STAR–machine, any arc $(u, v)$ will be matched with a triple $< u, v, \delta(u, v) >$, and $\delta(u, v)$ will use $h$ bits. We choose the parameter $h$ as the number of bits for coding $\sum_{i=1}^{n} c_i$, where $c_i$ is the maximum weight of arcs outgoing from the vertex $v_i$. Let us agree to represent a digraph $G$ as association of the matrices *Left*, *Right*, and *Cost*, where every arc $(u, v)$ occupies an individual row, $u \in Left$, $v \in Right$, and $\delta(u, v) \in Cost$.

To design an associative parallel algorithm for finding the second simple shortest paths, we will use the following data structure:

- an $m \times h$ matrix $Cost$ that saves the value of the function $\delta(u, v)$ for every arc $(u, v)$;
- an association of the matrices *Left*, *Right*, and $Cost$;
- a slice $Tree$, where *positions* of arcs belonging to the shortest paths tree are marked with the bit $'1'$;
- an $n \times h$ matrix $Dist$, whose every $i$-th row saves the distance from the root $v_1$ to the vertex $v_i$;
- an $m \times n$ matrix $TPaths$, whose every $i$-th column saves with the bit $'1'$ the positions of arcs that belong to the shortest path from $v_1$ to the vertex $v_i$;
- an $n \times \log n$ matrix $Code$, whose every $i$-th row saves the binary representation of the vertex $v_i$.

Let $P(r)$ denote the shortest path from $v_1$ to the vertex $r$ and $P(j, l)$ denote the shortest path from $j$ to $l$ in the given tree. Obviously, on the STAR–machine, $P(r)$=COL$(r, TPaths)$.

To obtain the matrix $Dist$ and the slice $Tree$, we propose a new implementation of the Dijkstra algorithm [1] on the STAR–machine. It is obtained by means of minor changes in the procedure DistPath for simultaneous finding the distances and the shortest paths on the STAR–machine [9]. More precisely, we include the matrices $Left$ and $Right$ as input parameters of the new procedure TreeDist and the matrix $Dist$ and the slice $Tree$ as its output parameters. As soon as we determine the current vertex $v_k$ for including into the shortest paths tree and a vertex $v_i$ being next to $v_k$, we will define the *position* of the arc $(v_i, v_k)$ in the association of matrices $Left$ and $Right$. After that, we include this position into the slice $Tree$. The procedure TreeDist takes $O(hn)$ time.

To obtain the matrix $TPaths$, we perform the procedure TreePaths that uses the following idea proposed in [10]. Assume we know *positions* of arcs included into $P(l)$. Then we construct a tree path for such a vertex $v_k$ which is the head of the arc $(v_l, v_k)$ in the shortest paths tree, and $P(k)$ has not been defined yet. The shortest path $P(k)$ is obtained by including the position of the arc $(v_l, v_k)$ into $P(l)$. The procedure TreePaths takes $O(n \log n)$ time.

To obtain the matrix $Cost$, we perform the procedure Recount that computes for every arc $(u, v)$ the function $\delta(u, v)$ considered in the previous section. The procedure Recount takes $O(r \log n)$ time, where $r$ is the number of non–tree edges.

We will use the following two properties of the matrix $TPaths$.

Property 1. Every $i$-th row of the matrix $TPaths$ saves the positions of vertices whose shortest paths from the root include the arc, which is written in the $i$-th row of the graph representation.

Property 2. If the shortest path from the root to the vertex $j$ is the prefix of the shortest path from the root to the vertex $l$, then $P(j, l) = P(l)\, and\, (\, not\, P(j))$.

## 5 Finding the Second Simple Shortest Paths

In this section, we will first explain the main idea of the associative algorithm for finding the second simple shortest paths from the root $v_1$ to all vertices of $G$. It

determines the *position* of the current non–tree edge $\alpha$ with the minimum weight in the matrix $Cost$ whose head has not been updated. Knowing the shortest path from $v_1$ to the tail of $\alpha$ and the *position* of the non–tree edge $\alpha$, the algorithm determines the second simple shortest path from $v_1$ to the head of $\alpha$. Further, it determines the second simple shortest path from $v_1$ to every vertex that belongs to the subtree rooted at the head of $\alpha$.

We first explain how to obtain all vertices from the subtree rooted at the head of $\alpha$, say $j$. Using Property 1 of the matrix $TPaths$, we can do this as follows. We determine *positions* of arcs outgoing from $j$ in the graph representation. Then for every such an arc, we determine all vertices, whose shortest paths from $v_1$ include it. The subtree rooted at $j$ is obtained as a set of all such vertices.

Now let us explain how to determine the second simple shortest path from $v_1$ to every vertex $q$ from the subtree, for which the second shortest path has not been obtained. Using Property 2 of the matrix $TPaths$, we can do this as follows. We include *positions* of arcs from $P(j, q)$ into the second simple shortest path from $v_1$ to $j$.

The associative algorithm for finding the subtree of the shortest paths rooted at $j$ performs the following steps.

Step 1. By means of a slice (say, $Z$), save *positions* of arcs outgoing from $j$ in the graph representation.

Step 2. While $Z \neq \emptyset$, update the arcs outgoing from $j$ as follows:

– select the *position* $l$ of the uppermost arc marked with the bit $'1'$ in the slice $Z$. Then set $'0'$ into the $l$-th bit of the slice $Z$;

– save the $l$-th row of the matrix $TPaths$;

– accumulate positions of these vertices.

On the STAR–machine, this algorithm is implemented as auxiliary procedure Subtree. Knowing the matrices $Left$, $TPaths$, and $Code$, the slice $Tree$ and the vertex $j$, the procedure returns a variable $v$ of the type word that saves the vertices belonging to the subtree rooted at $j$. To determine *positions* of the tree arcs outgoing from $j$, we perform the statement `w:=ROW(j,Code)` and the basic procedure `MATCH(Left,Tree,w,Z)`. The procedure Subtree takes $O(\log n + deg^+(G))$ time because the basic procedure MATCH takes $O(\log n)$ time, the cycle for updating the arcs outgoing from $j$ (Step 2) is performed $O(deg^+(G))$ times, and inside this cycle, every operation takes $O(1)$ time.

The associative algorithm for finding the second simple shortest paths from $v_1$ to the vertices from the subtree rooted at $j$ performs the following steps.

Step 1. By means of a slice (say $Y$) save the *second* simple shortest path from $v_1$ to $j$. By means of another slice (say $Y1$) save the shortest path from $v_1$ to $j$.

Step 2. By means of a variable of the type word (say $w2$), save those vertices from the subtree rooted at $j$, for which the second simple shortest paths have not been built yet.

Step 3. While $w2 \neq \emptyset$, determine the second simple shortest path to every vertex, whose *position* belongs to $w2$, as follows:

– select the *position* $q$ of the leftmost bit $'1'$ in the row $w2$. Then set $'0'$ into the $q$-th bit of $w2$;

– by means of a slice (say, $Y2$), save *positions* of arcs that belong to $P(j, q)$;

– include into the slice $Y2$ the *positions* of arcs that belong to the slice $Y$. Write the slice $Y2$ into the $q$-th column of the matrix $TPaths[2]$;

– in the $q$-th row of the matrix $Dist[2]$, write the weight of the second simple shortest path from $v_1$ to $j$.

On the STAR–machine, this algorithm is implemented as auxiliary procedure UpdateSubtree. Knowing the matrix $TPaths$ and the vertex $j$, it returns the matrices $TPaths[2]$ and $Dist[2]$. Initially, the $j$-th column of $TPaths[2]$ saves the second simple shortest path from $v_1$ to $j$, the $j$-th row of the matrix $Dist[2]$ saves the weight of this path, and the variable $w2$ saves vertices from the subtree, for which the second simple shortest paths have not been determined. The procedure UpdateSubtree takes $O(deg^+(G))$ time because the operations take $O(1)$ time each, and the cycle for updating vertices from the subtree (Step 3) is performed $O(deg^+(G))$ times.

Now we proceed to the associative algorithm for finding the second simple shortest paths from $v_1$ to all vertices of $G$. It performs the following steps.

Step 1. Let a slice (say, $Z1$) save *positions* of non-tree arcs in the graph representation. While $Z1 \neq \emptyset$, update the non–tree edges as follows.

Step 2. Determine the *position* $l$ of a non–tree edge $\gamma$ that has the minimum weight in the matrix $Cost$ and the second simple shortest path from $v_1$ to its head has not been constructed. Let $\gamma = (i, j)$.

Step 3. Build the second simple shortest path from $v_1$ to $j$ using $P(i)$ and the *position* $l$ of the non–tree edge $(i, j)$. Write this path into the $j$-th column of the matrix $TPaths[2]$. Write the weight of the arc $(i, j)$ into the $j$-th row of the matrix $Dist[2]$. Then mark the vertex $j$ as updated one.

Step 4. Determine the vertices from the subtree of the shortest paths rooted at $j$. By means of a variable of the type word (say, $w2$), save those vertices from the subtree, for which the second shortest paths have not been built.

Step 5. Build the second simple shortest path from $v_1$ to every vertex of the subtree, whose position belongs to $w2$.

On the STAR–machine, this algorithm is implemented as the main procedure SecondPaths that uses the auxiliary procedures Subtree and UpdateSubtree.

## 6 Implementing the Algorithm for Finding the Second Simple Shortest Paths

In this section, we consider the procedure SecondPaths. Knowing the graph representation as a list of triples, the matrix $TPaths$ and the slice $Tree$, it returns the matrices $TPaths[2]$ and $Dist[2]$ and a slice $Z$.

```
procedure SecondPaths(Left,Right:table; Cost:table; Code:table;
  TPaths:table; Tree:slice(Left); var TPaths[2]:table;
  var Dist[2]:table; var Z:slice(Left));
var i,j,l,q:integer;
```

```
   w: word(Code); u: word(Cost); w1,w2: word(TPaths);
   Y,Y1,Y2,Z1,Z2: slice(Left); X,X1: slice(Code);
 1. Begin SET(X); SET(w1);
 /* By means of w1, we save the vertices for which
   the second shortest paths have not been built. */
 2.  Z1:=not Tree; Z:=not Tree;
 3.  while SOME(Z1) do
 /* The cycle for updating non-tree edges. */
 4.    begin MIN(Cost,Z1,Z2); l:=FND(Z2);
 /* We determine the position of the non-tree edge having
   the minimum weight in the matrix Cost. */
 5.      Z1(l):='0';
 6.      w:=ROW(l,Left); MATCH(Code,X,w,X1);
 7.      i:=FND(X1);
 8.      w:=ROW(l,Right); MATCH(Code,X,w,X1);
 9.      j:=FND(X1);
 /* The non-tree edge (i,j) has been written in the l-th
   row of the graph representation. */
10.      if w1(j)='1' then
 /* The case when the vertex j has not been updated. */
11.        begin Y:=COL(i,TPaths); Y(l):='1'
 /* We include the position of the edge (i,j) into P(i). */
12.          COL(j,TPaths[2]):=Y;
13.          u:=ROW(l,Cost);
14.          ROW(j,Dist[2]):=u;
 /* The weight of the second shortest path to j is written
   in the j-th row of the matrix Dist[2]. */
15.          w1(j):='0';
 /* In the row w1, we mark the vertex j as updated one. */
16.          Z(l):='0';
 /* In the slice Z, we mark the arc (i,j) as updated one. */
17.          Subtree(Left,TPaths,Code,Tree,j,w2);
18.          w2:=w2 and w1;
 /* The row w2 saves the vertices for which the second
   simple shortest paths will be constructed. */
19.          w1:=w1 and (not w2);
20.          UpdateSubtree(TPaths,j,TPaths[2],Dist[2],w2);
21.        end;
22.    end;
23. End;
```

**Theorem.** *Let the matrices Left, Right, Cost, TPaths, and Code and the slice Tree be given. Then the procedure* SecondPaths *returns the matrices TPaths[2] and Dist[2], and a slice Z to save the positions of non–tree edges not used in building TPaths[2].*

**Proof (Sketch).** We prove this by induction on the number of non–tree edges $t$ in the shortest paths tree.

Basis is proved for $t = 1$. After performing lines 1–9, we obtain that the non–tree arc $(i, j)$ from the $l$-th row of the matrix $Cost$ has the minimum weight. After performing lines 10–16, we determine the second simple shortest path from $v_1$ to $j$ and write it in the $j$-th column of the matrix $TPaths[2]$ (line 12). Then we write its weight in the $j$-th row of the matrix $Dist[2]$ (line 14). After performing lines 17–20, we determine the second simple shortest paths from $v_1$ to all vertices of the subtree except $j$. Since $Z1 = \emptyset$ (line 5), we go to the exit.

Step of induction. Let the assertion be true when $t \geq 1$ non-tree edges are updated in the shortest paths tree. We will prove this for $t + 1$ non–tree edges.

By the inductive assumption, after updating $t$ non–tree edges, their positions are marked with $'0'$ in the slice $Z1$, the slice $Z$ saves positions of non–tree edges not used for finding the second simple shortest paths, the matrix $TPaths[2]$ saves the second simple shortest paths for vertices, whose positions are marked with $'0'$ in the row $w1$, and the matrix $Dist[2]$ saves the weights of the corresponding paths. Now we update the last non–tree edge whose position is marked with $'1'$ in the slice $Z1$. By analogy with the *basis*, after performing lines 4–9, $Z1 = \emptyset$ and we determine that the non–tree edge $(i, j)$ is written in the $l$-th row of the graph representation. Now, we perform line 10. If $w1(j) =' 1'$, we reason as in the case of the *basis*. Otherwise, after line 10, we go to the exit, and the position of this non–tree edge will be marked with $'1'$ in the slice $Z$. In this case, the second simple shortest path from $v_1$ to $j$ (say $\alpha$) has been built before and it uses a non–tree edge, whose weight is less than $\delta(i, j)$. Therefore, the path $\alpha$ is really the second shortest path from $v_1$ to $j$.

Let us evaluate the time complexity of this procedure. Let $r$ be the number of non–tree edges in the shortest paths tree that are really used for finding the second simple shortest paths. Then the procedure SecondPaths takes $O(r(\log n + deg^+(G)))$ time because in the cycle for updating the non–tree edges, the basic procedures MIN and MATCH take $O(\log n)$ time each, the auxiliary procedure Subtree takes $O(\log n + deg^+(G))$ time, the auxiliary procedure UpdateSubtree takes $O(deg^+(G))$ time, and other elementary operations of the STAR–machine take $O(1)$ time each.

This completes the proof.

## 7 Conclusions

We have proposed the efficient associative algorithm for finding the second simple shortest paths from the source vertex $v_1$ to all vertices of a digraph. This algorithm uses the data structure that allows us to extract data by contents. On the one hand, the data structure uses a special implementation of the classical Dijkstra algorithm on the STAR–machine [10]. On the other hand, it uses the function $\delta(u, v)$ proposed by Eppstein [2]. The associative algorithm for finding the second simple shortest paths is implemented on the STAR–machine as procedure SecondPaths, whose correctness has been proved. The procedure takes

$O(r(\log n + deg^+(G)))$ time, where $r$ is the number of non–tree edges that are really used for finding the second simple shortest paths.

To find the second simple shortest paths, Roditty and Zwick [11] apply their efficient algorithm for finding the replacement paths, while Gotthilf and Lewenstein [4] apply the efficient algorithm of Pettie for finding the all-pairs shortest paths. Our technique uses the main properties of the associative parallel processors.

We are planning to study the representation of the replacement paths problem on the STAR–machine.

## References


1. Dijkstra, E.W.: A Note on Two Problems in Connection with Graphs. Numerische Mathematik 1, 269–271 (1959)
2. Eppstein, D.: Finding the k Shortest Paths. SIAM J. of Computing 28, 652–673 (1998)
3. Foster, C.C.: Content Addressable Parallel Processors. Van Nostrand Reinhold Company, New York (1976)
4. Gotthilf, Z., Lewenstein, M.: Improved Algorithms for the $k$ Simple Shortest Paths and the Replacement Paths Problems. Information Processing Letters 109, 352–355 (2009)
5. Katoh, N., Ibaraki, T., Mine, H.: An Efficient Algorithm for $k$ Shortest Simple Paths. Networks 12, 411–427 (1982)
6. Lawler, E.L.: A Procedure for Computing the $k$ Best Solutions to Discrete Optimization Problems and its Application to the Shortest Path Problem. Management Science 18, 401–405 (1972)
7. Nepomniaschaya, A.S.: Language STAR for Associative and Parallel Computation with Vertical Data Processing. In: Mirenkov, N. (ed.) Proc. of the Intern. Conf. "Parallel Computing Technologies, pp. 258–265. World Scientific, Singapore (1991)
8. Nepomniaschaya, A.S., Dvoskina, M.A.: A Simple Implementation of Dijkstra's Shortest Path Algorithm on Associative Parallel Processors. Fundamenta Informaticae 43, 227–243 (2000)
9. Nepomniaschaya, A.S.: Simultaneous Finding the Shortest Paths and Distances in Directed Graphs Using Associative Parallel Processors. In: Proc. of the Intern. Conf. "Information Visualization", IV 2003, pp. 665–670. IEEE Computer Society, Los Alamitos (2003)
10. Nepomniaschaya, A.S.: Associative Parallel Algorithms for Computing Functions Defined on Paths in Trees. In: Proc. of the Intern. Conf. on Parallel Computing in Electrical Engineering, PARELEC 2002, pp. 399–404. IEEE Computer Society, Los Alamitos (2002)
11. Roditty, L., Zwick, U.: Replacement paths and $k$ simple shortest paths in unweighted directed graphs. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 249–260. Springer, Heidelberg (2005)
12. Yen, J.Y.: Finding the k Shortest Loopless Paths in a Network. Management Science 17(11), 712–716 (1971)

# The Construction of Simulation Models of Algorithms and Structures with Fine-Grain Parallelism in WinALT

Mike Ostapkevich and Sergey Piskunov

The Institute of Computational Mathematics and Mathematical Geophysics, SB RAS,
Lavrientieva 6,
630090 Novosibirsk, Russia
{ostap,piskunov}@ssd.sscc.ru

**Abstract.** The user features of WinALT system are discussed. It is demonstrated that WinALT is a convenient tool for the construction of simulation models of fine-grain algorithms and structures. It is a tool that combines graphical and analytical means and is adaptable to needs of a particular problem domain. The open modular architecture of the system lets a user actively participate in the extension of system's functionality. Representative samples of simulation models are presented. The availability of the system in Internet makes it possible to extend the range of its application by joint efforts of users and developers.

**Keywords:** fine-grain parallelism, simulating system, open software architecture, component based architecture, user friendly interface, samples of simulating models.

## 1 Introduction

There is currently a tendency to create software with open access. The extension of its functionality can be done not only by its developers, but also by its users.

The *objective* of this work is to demonstrate the capabilities of the WinALT system available at the site [1] in constructing of models of algorithms and structures with fine-grain parallelism in order to attract potential users to the development of the system in the directions that they are interested in within the domain of computations and architectures with fine-grain parallelism. This domain includes not only cellular automata with its extensions, but also matrix and pipeline architectures (including systolic), multimicroprocessor architectures, associative processors, cellular-neural networks, homogeneous reconfigurable computing systems (including FPGA), etc.

## 2 Custom Features of WinALT System

The system has a number of features that are useful for its adjustments to users' needs and for a comfortable construction of models by authors' opinion:

- free distribution,
- common user's interface,

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 192–203, 2011.

- simulating language that combines the capabilities of parallel and sequential programming and that is able to adjust to users' requirements,
- graphical capabilities that turn the system into a visual programming environment,
- collection of models at the system's site,
- extensible and modular system architecture that is based on components,
- system implementation in C/C++ languages,
- points of functionality extension, which are available to users.

**Free Distribution.** The system's site [1] contains "system installation" section that includes an installation and operation manual and its distributive package.

**Common User's Interface.** The user's interface of the system coincides with the standard user's interface in Windows applications. A simulation model is represented by a project that contains a number of sub-windows. Each sub-window can hold graphical or textual objects of a model. Creation and editing of graphical objects is performed by the means of toolbars, menus and dialog windows. Dialog windows have a layout that is well known for Windows users. Texts of model programs are created and modified by a system's editor that has functionality similar to that of Notepad. The main features of user's interface of the system remain unchanged since its first presentation [2].

### 2.1 Adequacy of the System's Language to the Problem Domain

**General Description of Simulating Language.** Samples in the selected problem domain of fine-grain computations may have such deviations from a cellular automata as heterogeneity of cellular space, global data dependencies between cells (e.g. buses), block structure, centralized or distributed coordination of inter-block interactions, wide range of cell complexity (from a cell with a small transition table to a cell as complex as a microprocessor), etc. The system's simulating language pushes the limits of the representation of classic cellular automata by its capability to describe the above mentioned abnormalities. It consists of three parts. First, their general description will be presented. Then, the part dedicated to the description of parallel computations will be examined in details. And finally, the general structure of a model program will be presented. Let us note that the samples considered in the section 3 elucidate quite exhaustively the main concepts of this section.

The first part of the language is designed to describe parallel computations in a form of parallel substitutions. It is fully based on an algorithmic system named Parallel Substitution Algorithm [3]. The selection of this algorithmic scheme is determined by the fact that it worked well in describing informational and physical processes in a wide variety of fine-grain architectures [4, 5, 6, 7].

The objective of the second part of the language is the description of sequential computations. This part is essentially based on Pascal. It provides statements for the description of program structure, control operators, assignment operator and subroutine call by name or by reference. These statements can be used in a model program for the description of sequential control when it is necessary. These means can be used in a model program for the description of sequential control, when it is needed, for the definition of functions that describe cell states and also for the

construction of such service functions within model program as menu definition, graph drawing or initial data input.

The third part of the language provides import of libraries into a model program. These are *dll* libraries written in C/C++ and embedded into the simulating system. They help to extend the functionality of system to a direction desirable for a user.

**The Description of the Parallel Part of Simulating Language.** This part has a clear division into graphical and analytical subparts. For clarity, a two-dimensional case will be considered. Graphical objects are cellular arrays and templates. The rectangular matrix of finite dimensions that is composed by colored cells and located along horizontal ($x$) and vertical ($y$) axes is called cellular array. The origin of coordinates is located at the top left cell of array. The cell coordinates *[x,y]* mean that there are *x-1* cells between this cell and the origin along $x$ axis and *y-1* cells along $y$ axis. A color is used to visualize cell's state. Its state can belong to any cell's data type supported by the library of data formats (considered in section 2.2 in depth). Each cellular has a name. A template can differ from a cellular array. There can be cells in void state in it. Such a cell is depicted by a diagonal cross. Some of the cells can be marked. A name is assigned to each such cell. This name is available for viewing and editing in graphical representation of a cell. Let us call the origin in template its "center".

The main operators of this part of the language are a parallel substitution composite *in-at-do* operator and an *ex-end* composite operator named synchroblock. The *in-at-do* substitutions are divided into two classes: symbolic and functional. The symbolic substitution is one that doesn't use templates with marked cells. If there is at least one template with at least one marked cell, the substitution is called functional. Each of these classes is divided into two subclasses in its turn: simple and vector substitutions. Simple substitutions perform data transformations in only one cellular array, while the vector ones affect more than one cellular array.

**The Structure and Execution of a Simple Symbolic Substitution.** *in* operator contains parameter name $W$ of processed cellular array. The parameter in *at* operator is name $L$ of the template of the left part of substitution. Similarly, the parameter of *do* operator is the name $R$ of the template of the right part of substitution. There is only one limitation for such a substitution. The template $R$ cannot have greater dimensions than $L$ along axes $x$ and $y$. One iteration of application of substitution to the cellular array $W$ is performed in two stages. At the first stage all occurrences of $L$ in $W$ are marked up to void cells while moving its center in $W$ along axes $x$ and $y$. At the second stage the cell states of $W$, which are within all the found occurrences, are simultaneously changed to the cell states of $R$ also up to void cells.

*Remark 1.* The symbolic substitutions are usually used for transformations of cellular arrays with types *bit*, *byte*, *int32* and other types with integer cells.

**The Structure and Execution of a Simple Functional Substitution.** The parameter of *in* operator is the name $W$ of processed array. The parameter of *at* operator is the name template $L$ in the left part of substitution. The parameter of *do* operator is a function name $F$. At least one cell in $L$ is marked. Cell names in $L$ play role of local variables in $F$. One iteration of application of substitution to $W$ is performed in two stages. At the first stage the variables that correspond to values of cells in $L$ get new values that are the states of cells in $W$ underneath them at each occurrence of $L$ as its

center moves in *W* along the *x* and *y*. At the second stage cells that were marked by variables from the list of output variables in each occurrence of *L* in *W* simultaneously get new values computed in *F*. The complexity of *F* is not limited. *F* is implemented with the help of the second part of the simulating language.

*Remark 2.* Template *L* moved with step size 1 along the axes *x* and *y* in the both discussed kinds of substitution. The size of step can be altered by the parameters of *step* operator that can be placed before *at* operator. The introduction of *on* operator after *in* operator limits an area of substitution applicability in *W* by a rectangular region with origin and sizes specified as parameters of *on* operator.

**The Structure and Execution of a Vector Symbolic Substitution.** The parameters in operators *in*, *at*, *do* of this kind of substitution are lists of names rather than names. A mode of coordinated movement of templates of the left part in their corresponding cellular arrays and of coordinated search of their occurrences and alteration of cell states in cellular arrays for the cells that correspond to the templates of the right part is implemented in the language.

**The Structure and Execution of a Vector Functional Substitution.** The structure of the left part and the mode of template movement and occurrence search are similar to those for the vector symbolic substitution. Unlike simple functional substitution, the function set as a parameter in *do* operator can use cell names not from one cellular array, but from any subset of arrays from the list in *at* operator.

***ex - end* Synchroblock.** A sequence of operators of substitutions of any kind is placed between *ex* and *end* operators. The synchroblock sets an iterative application of substitutions. One iteration consists of ubiquitous application of substitutions to processed cellular arrays that are specified by names in a correspondent *in*. The iterations are repeated while at least one is applicable.

*Remark 3.* Besides *ex - end* synchroblock there are two specialized kinds of synchroblock as well: *ch – end* synchroblock executes its substitution only once, and *cl – end* synchroblock executes its substitutions a number of iterations specified in the parameter of *cl* operator.

*Remark 4.* Let us note that only the most typical operators of the parallel part of the language are described above. Particularly, 1D, 2D, 3D arrays of various sizes and with versatile types of cell values can be used in simple substitutions as well as miscellaneous combinations of these arrays can be used in vector substitutions. The operator of synchronous assignment *let* can be used as well in synchroblocks. The neighborhood for a cell with coordinates *[x, y]* is defined by a set of functions with arguments *[x, y]* rather than by a template. This permits to define non-local neighborhoods. The functions of the set are defined with the help of the second part of the language. Operator *let* doesn't have a graphical image.

*Remark 5.* The composition of parallel and sequential parts of the language is achieved by the ability to use sequential operators in synchroblocks.

**Simulating Program.** The structure of a model program is rather traditional. It consists of a list of libraries imported to program, declarations of constants, variable and cellular objects, procedures, functions and the main operator block. The main operator block is placed in operator *begin-end* brackets and contains operators of the first and second parts of the language. A program may include a comment, which can

be any text placed in braces. A project can contain any number of simulating programs. The system's interface allows a user to activate a specified simulating program out of many within a project by an execution menu. It is also possible to define a menu in one simulating program to select and invoke operations defined in another one within the same project. Each simulating program can have its own set of cellular objects or share some of its objects with other programs in the same project. Objects can also be shared between projects by placing them to a folder for global objects.

### 2.2 The Architecture of WinALT System

Development of the system follows the path of improving its modular architecture and the creation of tools for design and incorporation of new modules. A block diagram of the system is presented at Fig. 1.

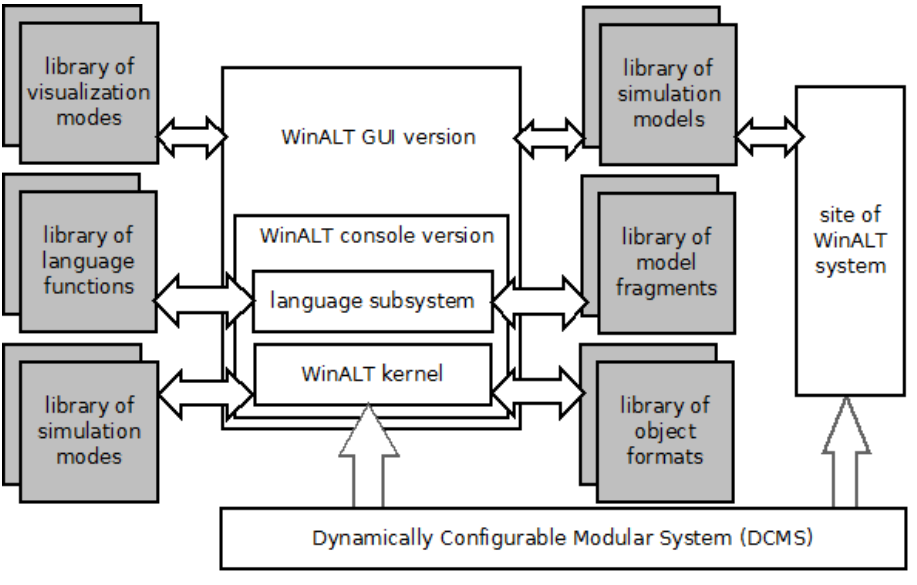



**Fig. 1.** Block diagram of WinALT system

The comparison of current system's architecture with its earlier version [2] reveals the following differences: A) a division of the system's kernel produced two components: the WinALT kernel and DCMS library; B) a set of libraries is formed in order to provide system's adaptability to requirements of versatile users' problem domains.

A) DCMS is a system library built above operating systems (Win32, Linux). It provides functions for management of data structures and for inter-modular communications based on events. The objective of DCMS is to provide openness of system not only at the level of adding new modules, but also at the level of assembling of new components on the basis of these modules. For example, the site of WinALT system is assembled from fragments (HTML and SVG generators,

authorization module, etc.) using DCMS library. The sections dedicated to the description of WinALT system including its distributive and such issues as how to build, execute and debug simulation models in it as well as a collection of such models, references on cellular computations, the description of existing simulating systems are presented at the site. Currently a parallel version of the WinALT system based on DCMS is under development.

The WinALT kernel uses DCMS functions for processing of data structures (atomic and hierarchical data built using such constructors as array and record), associative search by key and event generation. The WinALT kernel implements in its turn functions that are specific for the problem domain of fine-grain computations and architectures. These functions manage cellular arrays and templates, provide fast access to them, conduct search of applicable substitutions and simulate their parallel execution. The console version is capable to execute simulation models, but it lacks a friendly user's interface. The graphical environment compensates this limitation providing mature GUI tools for visual composition and debugging of models. The system is implemented in C/C++. The stable version of graphical environment is based on MFC[8]. The extraction of stand-alone console version makes it possible to create alternative graphical environments. For example, an addition of graphics based on Trolltech QT [9] and GNU GTK [10] into the system can be done.

B) Each library is refilled by new modules. The libraries are composed of external modules. The typical case is when such a module is a *dll* written in *C/C++*. Its inclusion into library is automated by the tools of DCMS library. To include a module means to make its functions callable from any simulating program of WinALT system. As these tools of inclusion are currently insufficiently documented, they can be applied by users only with the help of developers.

Lets us describe briefly the functionality of libraries.

*The library of data formats* eliminates limitations of data type that can be represented by cells in cellular array. The library contains modules for representations of cellular arrays with integer cells (*int8, int16, int32, uint8, uint16, uint32*), bit cells (*bit*), float cells (*float*) and others. Some external formats are supported by the modules of library, such as *bmp* raster graphics format. The assignment of *default* type for a cellular object means that any of its cells can have any of the above mentioned formats. That can be used for the representation of heterogeneous cellular objects. In GUI a type of cellular object can be selected in a combo box within the dialog window of new object creation.

*The library of language functions* provides the ability to use such functions in simulating programs as functions of object management (creation, deletion, modification or size alteration), GUI functions (construction of dialog windows and data input based upon them), mathematical functions (*sin, cos, atan, cosh, log, j0*), console I/O functions *(WriteLn, ReadLn),* file I/O functions *(fopen, fgets, fread, feof, …)* and miscellaneous functions such as *max, min, null, typeof, StringLength, Time*. Operator *use* activates the modules of this library in a simulating program.

*The library of model subroutines* provides inclusion of service functions into a simulating program by a preprocessor command *include*. Modules of the library are written in WinALT language. The module *plot.inc* implements drawing of graphics, while the module *dcmc_io.inc* provides functions for user's I/O based on dialog windows that is typically used for such purposes as selection of initial configuration

in model, assignment of object name or setting of number of steps of model execution.

*The library of visualization modes* provides the ability to visualize data in a form that is familiar to a user or widely accepted in his problem domain. For example, module *num.1.dll* shows numeric values of cell. Module *arrow.1.dll* visualizes cell values as arrows with a certain direction. Module *cont.1.dll* draws contours around cell areas. Module *rect.1.dll* draws a rectangular grid of cells, while *hex.1.dll* draws a hexagonal grid. The visualization modes are selected from menu.

*The library of simulation modes* supports three modes (synchronous, asynchronous and block asynchronous) selectable be user. In the synchronous mode all the applicable substitutions are executed at each iteration of processing of cellular arrays. It is this mode that is chosen in the description of the first part of the simulating language. In the asynchronous mode only one randomly selected applicable substitution is executed at each step. In the block asynchronous mode a certain number of applicable substitutions is executed. New modes can be added to the library. A simulating mode can be selected by user in menu.

*The library of simulating models* is transferred to a user together with the system distributive. It can serve as a basis of his personal model library that can be extended by own models or ones from WinALT site or from other sources. The library is kept in a dedicated directory.

## 3 Samples of WinALT Models

The selected models are quite versatile. The first model contains various 2D cellular objects, while the second one has 3D cellular objects. There are bit cell states in the first model. The states of cells in the second model are float. The text windows of the first model provide a visual representation of the structure of simulating program. Nested synchroblocks are used in simulating programs of both samples as well as the calls of modules of system libraries. In the first model the substitutions number 1, 2, 3, 4 are vector functional ones, while the substitution 5 is simple symbolic one. The only substitution of the second model is a simple functional one. The second model contains a menu created by developers using the second part of WinALT language.

### 3.1 A Simulation Model of Associative Device

**A Search Algorithm of Binary Table Strings that Match with a Reference Pattern.** In order to mark strings coinciding with a reference pattern a binary register is introduced. Its bit length is equal to the number of rows in the table. All its digits are set to *TRUE* in the initial state. Let us enumerate the digits of the reference pattern as well as the table columns from left to right by *0, 1, …, x, ….* At each *x-th* iteration of algorithm execution the *x-th* digit of the reference pattern is selected and simultaneously compared with all the digits of the *x-th* column. Let a certain digit unmatched with its respective bit in the reference pattern be in a *y-th* row of the table. If the *y-th* digit of the register contains the *TRUE* symbol, it is replaced by the *FALSE* symbol. The search ends when all the digits of the reference pattern are processed. The strings coinciding with the reference pattern appear to be marked by the *TRUE* symbol in the register.

**A Description of Model of the Associative Device.** A screenshot of the model project is shown at Fig. 2. The final iteration of model execution is presented there.

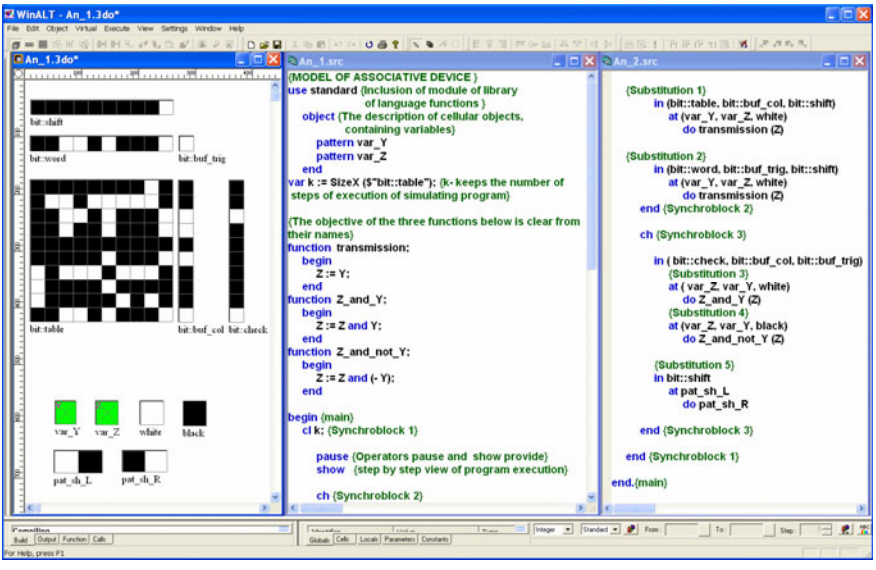



**Fig. 2.** A project of model of an associative device

All the graphical objects of model are shown in *An_1.3do* window. *TRUE* state of a cell is represented by white color, while *FALSE* state is represented by black color. The cellular arrays are used for storing of model data as follows: a) *bit::table* for binary strings of the table, b) *bit::word* for reference pattern, c) *bit::check* for the results of comparison, d) *bit::shift* for a marker (a white cell), e) *bit::buf_col* for the next (*x-th*) column of the table, f) *bit::buf_trig* for the next (*x-th*) digit of the reference pattern. In the initial state the marker is written to the left-most cell of *bit::shift* array, all the cells of *bit::check* array are white, while all the cells of *bit::buf_col* and *bit::buf_trig* arrays are black. All the templates have *default* type. Each of *var_Y* and *var_Z* templates consists of one empty cell. The cell of *var_Y* template is named as *Y*, while the cell of *var_Z* template is named as *Z*. These templates are used in functional substitution of the model program. The *pat_sh_L* and *pat_sh_R* templates are used in the substitution command that implements shift of the marker along the cells of *bit::shift* array. The templates white and black are used in functional substitution to set the conditions of applicability.

The simulating program is presented in *An_1.src* and *An_2.src* windows. After its launch the size of *bit::table* array along axis *x* is calculated by the *SizeX* library function and is assigned to the variable *k*. This variable sets the number of steps in the synchroblock 1. Each step consists of two sequentially executed iterations. The first iteration is set by the synchroblock 2, and the second one is set by the synchroblock 3.

*The first iteration.* The substitution 1 using *transmission* function copies the column that is marked in *bit::shift* from *bit::table* to *bit::buf_col*. The substitution 2

using the same function copies the reference pattern bit marked also by *bit::shift* from *bit::word* to *bit::buf_trig*.

*The second iteration.* The substitution 3 alters white cell by a black one in *bit::check* for each pair of cells in *bit::check* and *bit::col* with the same number if the cell in *bit::buf_col* is black and the cell *bit::buf_trig* is white using the function *Z_and_Y*. The substitution 4 alters white cell by a black one in *bit::check* for each pair of cells in *bit::check* and *bit::col* with the same number if the cell in *bit::buf_col* is white and the cell *bit::buf_trig* is white using the function *Z_and_not_Y*. The substitution 5 shifts the marker to the right by one cell in *bit::shift*.

*Remark 6.* The substitutions 4 and 5 have the same *in* operator, and in such a case it can be shared.

The white cells in *bit::check* denote the two strings in *bit::table* that match with the reference pattern at Fig. 2.

### 3.2 A Simulation Model of Algebraic Fractal "Julia Set"

**The Algorithm of Construction of Fractal "Julia Set"**[11]. The algorithm is based on iterative expression:

$$\boldsymbol{Z}_{i+1} = \boldsymbol{Z}_i^{\,2} + \boldsymbol{C} \tag{1}$$

where $Z_i$ and $C$ are complex variables. The iterations are performed for each starting point $Z$ of a rectangular or square region, a subset of complex plane. The iterative process goes on until $Z_i$ goes beyond a circle of radius 2 with the center at the point (0,0). In this case the point takes white color (background color) or after a sufficiently big number of iterations $Z_i$ sequence converges to some point of the circle (the point takes red color).

**The Description of WinALT Model of Fractal Variant Construction.** A model screenshot is depicted at Fig. 3. The final iteration of model execution and a menu for selection of the next variant to construct are presented there. The cellular objects of the model are depicted in the left window:

a) *float::area* is a 3D cellular array (with three layers) with the size 350x350x3 cell. Its dimensions along axes *x* and *y* can be changed arbitrarily using the simulating system tools taking computational capabilities of the computer into consideration. Axis *z* is directed away from the viewer into the screen. A user can observe any of the layers or all the three of them unrolled in a plane as shown at Fig. 3. Initially all the cells in the layer 0 are red colored (have value *1.000000*). A complex number $Z_{xy}$ is associated with each column of the two cells of layers 1 and 2 along axis *z*. Its real part $Re(Z_{xy})$ is the state of cell of the layer 1, and the imaginary part $Im(Z_{xy})$ is the state of cell of layer 2.

b) *tpl* is a template for the main functional substitution that computes whether a cell belongs to Julia set. Its dimensions are 1x1x3 cells. It is shown in an unrolled form at the Fig. 3. The template cell's name in the layer 0 is set to be *fr*, while those in the layers 1 and 2 are *rl* and *im* respectively. The template's type is *default*. The cell type of layer 0 is *float* and its value is *1.000000* (red color). The type of cells in layers 1 and 2 is *void* (they have no value).

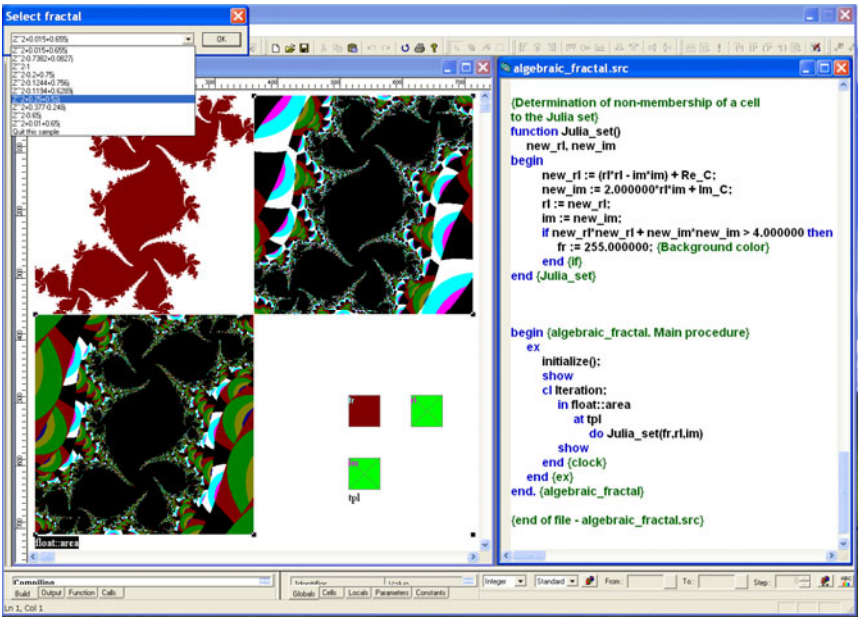



**Fig. 3.** A project of model of algebraic fractal

The structure of simulating program is similar to that in the previous sample, thus its detailed description is skipped. Let us note that two ACL libraries (*standard*, *altio*) and a model fragment (*dcms_io*) are included into this program. It contains the following functions and procedures: *set_c(rea, ima)*, *set_xy()*, *initialize()*, *Julia_set()*. They are written only with the operators of the second part of the language. The final part of simulating program, including function *Julia_set()* and the main program block is presented in window *algebraic_fractal.src* at the Fig. 3.

Let us consider the execution of the simulating program. It starts with the launch of procedure *initialize()* in the *ex-end* synchroblock. The procedure constructs menu *"Select fractal"* (see Fig. 3). A user has a choice of 10 variants of fractal set by expressions. After making a choice, a user presses *OK* button and the procedure *set_xy()* is called from the procedure *initialize()*.

```
{The initialization of cellular array float::area}
procedure set_xy()
 x, y {declaration of local variables}
begin
 for y := 0 to SIZE_Y-1 do
  for x := 0 to SIZE_X-1 do
   float::area(x, y, 0) := 1.0; {1st operator}
   float::area(x,y,1):=(2.*x)/SIZE_X-1.; {2nd operator}
   float::area(x,y,2):=(2.*y)/SIZE_Y-1.; {3rd operator}
  end {for}
 end {for}
end {set_xy}
```

*SIZE_X* and *SIZE_Y* define the dimensions of *float::area* array along *X* and *Y* respectively. The first operator "paints" all the cells of the layer 0 to red color. The second operator initializes each cell with coordinates *[x, y, 2]* with a float value that is interpreted as $Re(Z_{xy})$. The third operator initializes each cell with coordinates *[x, y, 1]* with a float value that is interpreted as $Im(Z_{xy})$. As a result complex values in the range from (-1., -1.) to (1., 1.) are presented in the layers 1 and 2 of array *float::area*. The procedure *initialize()* calls procedure *set_c(rea, ima)* that initialized variables *Re_C* and *Im_C* with parameters from a formula that a user selected on each menu item except the last one.

Now synchroblock *cl iteration-end* nested in synchroblock *ex-end* is executed. The value of *iteration* is set by a user when defining constants. The functional substitution in this synchroblock is executed *iteration* times. Its application means that at the first stage of each iteration all the occurrences of *tpl* template are marked in array *float::area*. Each such occurrence contains a cell in layer 0 with the state coinciding with that of the cell in the layer 0 of *tpl*, i.e. the cell is red. The variables *rl* and *im* get values that are the states of cells with coordinates *[x, y, 1], [x, y, 2]* respectively for each application of template with red cell having coordinates *[x, y, 0]*. Using these values function *Julia_set()* computes new states of cells with coordinates *[x, y, 1], [x, y, 2]* and checks if a cell with coordinates *[x, y, 0]* belongs to the Julia set. At the second stage new states are written into the cells with coordinates *[x, y, 1], [x, y, 2]* with the help of variables *rl* and *im*. Also, if the cell with coordinates *[x, y, 0]* doesn't belong to the Julia set, its state is assigned to the background color (value 255.0000) using variable *fr*.

Using synchroblock *ex-end* in this simulating program means that a user can initiate its execution an unlimited number of times. In order to stop the execution, a user has to select the last string of the menu ("*Quit this sample*") and then to press *OK* button.

*Comment.* This model was developed by students E. Umrikhina and M. Romanetz of the Novosibirsk State Technical University under supervision of one of the authors of this paper as a part of learning of WinALT system.

## 4 Collection of Models at the System's Web Site

The system was tested in the construction of models for cellular automata, including 2D FHP models of a physical process, diffusion models; arithmetic devices, multistage 2D and 3D micropipes, including ones with dynamic reconfiguration; cellular-neural networks for recognition of distorted images; cellular models of Petri network; cellular models of geometric and algebraic fractals; artificial life and others. A small fraction of these models is published at the system's site using the same scheme of documentation. The collection is constantly enriched by new models.

## 5 Conclusion

The further work in order to improve usability will be conducted in the following directions:

- Finishing the development of component - parallel version of WinALT with construction of tools for Web access to the resources of supercomputers.
- The extension of visualization tools by constructing new components - graphical environments of the system based on GTK, QT and KDE libraries.
- The extension of functionality of the system's site in order to provide communications between users, publication of materials by users, addition of new models to collection, the use of models and their fragments to construct new models, addition of new modules and possibly components to the system.

## References

1. WinALT site, http://winalt.sscc.ru/
2. Beletkov, D., Ostapkevich, M., Piskunov, S., Zhileev, I.: WinALT, a software tool for fine-grain algorithms and structures synthesis and simulation. In: Malyshkin, V.E. (ed.) PaCT 1999. LNCS, vol. 1662, pp. 491–496. Springer, Heidelberg (1999)
3. Achasova, S., Bandman, O., Markova, V., Piskunov, S.: Parallel substitution algorithm. Theory and Application. World Scientific, Singapore (1994)
4. Tverdokhleb, P.E. (ed.): 3D laser technologies (2003) (in Russian)
5. Bandman, O.L.: Cellular-automata simulation of spatial dynamics. Informatics systems (10), 59–113 (2006) (in Russian)
6. Achasova, S.M.: Program constructor of cellular self-reproducing structures. Programming and Computer Software 35(4), 190–197 (2009)
7. Piskunov, S.V., Umrikhina, E.V.: Computer simulation of dynamically reconfigurable multistage micropipes. Scientific Service in Internet: Solution of big problems (September 22-27, 2008) (in Russian)
8. Shepherd, G., Kruglinski, D.: Programming with Visual C++.NET, 6th edn. Microsoft Press, Redmond (2003)
9. KDE site, http://www.kde.org/
10. GTK site, http://www.gtk.org/
11. Crownover, R.M.: Introduction to Fractals and Chaos in dynamic systems. Medieval & Renaissance Texts & Studies. Jones & Bartlett Publishers, MA (1996)

# Simulation of Heterogeneous Catalytic Reaction by Asynchronous Cellular Automata on Multicomputer*

Anastasia Sharifulina[1] and Vladimir Elokhin[2,3]

[1] ICM&MG SB RAS, Pr. Lavrentjeva, 6, Novosibirsk, Russia
[2] G.K. Boreskov Institute of Catalysis, SB RAS, Pr. Lavrentyeva, 5, Novosibirsk
[3] Novosibirsk State University, Pirogova 2, Novosibirsk, Russia
sharifulina@ssd.sscc.ru, elokhin@catalysis.ru

**Abstract.** In the paper parallel implementation of ACA simulating dynamics of carbon monoxide oxidation over the Pd(100) is presented. Parallel implementation of ACA is based on its approximation by block-synchronous CA. To estimate approximation accuracy comparative analysis of statistical characteristics and bifurcation diagrams, obtained by ACA and BCSA simulation, is performed. Results of parallel implementation of BSCA algorithm and estimations of its efficiency are presented.

**Keywords:** asynchronous cellular automata, block-synchronous mode, parallel implementation, experimental estimation, catalytic oxidation reaction.

## 1 Introduction

The heterogeneous catalysis plays important role in many areas of human activity. Therefore, studying of the mechanism of catalytic reactions is important both from fundamental and practical point of view. For computer simulations of heterogeneous catalytic reactions asynchronous cellular automata (ACA) with probabilistic transition rules being sometimes referred to as Monte-Carlo method are most suitable [1].

Computer simulation of chemical reactions on micro-level requires considerable computational capability. Therefore, efficient parallel algorithms are urgent need. In [2, 3] a method of efficient parallelization based on block-synchronous CA (BSCA) approximation of ACA is proposed. Meanwhile, approximation accuracy in general case is not studied. Hence, for each ACA class it is necessary to perform its assessment.

The aim of this article is twofold: 1) to show the validity of using BSCA for simulation of reaction-diffusion systems in the case of carbon monoxide (CO) oxidation reaction over the Pt(100), 2) to develop the algorithm of BSCA parallelization and obtain its efficiency by implementation on cluster.

* Supported by (1) Presidium of Russian Academy of Sciences, Basic Research Program N 14-6 (2011), (2) Siberian Branch of Russian Academy of Sciences, SBRAS Interdisciplinary Project 32 (2009), (3) Grant RFBR 11-01-00567a.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 204–209, 2011.

Catalytic CO oxidation reaction over the Pt(100) is studied by Monte-Carlo method in [4]. This reaction is of great intertest due to the oscillations caused by the reversible phase transition of the Pt(100) surface (hex $\leftrightarrow 1 \times 1$) induced by CO adsorption.

Apart from the Introduction and the Conclusion the paper contains two sections. In section 2 the CA-model of the reaction and simulating results are presented. In section 3 the ACA to BSCA transformation is described, results of computer experiments and efficiency estimations of parallel implementation of BSCA are given.

## 2 Carbon Monoxide Oxidation Reaction over Platinum Surface

### 2.1 CA Model of CO Oxidation Reaction over Pt

The mechanism of CO oxidation reaction over Pt(100) is described by 9 elementary stages presented in [4]. The ACA simulating the reaction is defined by three concepts [1, 2]: $\aleph_\alpha = \langle A, X, \Theta \rangle$, where $A$ is a cells *state alphabet*, $X$ is the *set of names*, $\Theta$ is *transition rule*.

The state alphabet is chosen according to the reagents participating in the reaction: $A = \{*_{1\times1}, *_{\text{hex}}, \text{CO}_{\text{ads}}^{1\times1}, \text{CO}_{\text{ads}}^{\text{hex}}, \text{O}_{\text{ads}}^{1\times1}, \text{O}_{\text{ads}}^{\text{hex}}\}$, where $*_{1\times1}$ and $*_{\text{hex}}$ denote the vacant active center of surface with square and hexagonal structure, $\text{CO}_{\text{ads}}^{1\times1}$ and $\text{CO}_{\text{ads}}^{\text{hex}}$ are carbon monoxide adsorbed on $1\times1$ and hex surface, $\text{O}_{\text{ads}}^{1\times1}$ and $\text{O}_{\text{ads}}^{\text{hex}}$ are oxygen adsorbed on $1 \times 1$ and hex surface.

The set of names $X = \{(i,j) : i = 1, \ldots, M_i, j = 1, \ldots, M_j\}$ is a set of integer cells coordinates in the discrete space corresponding to the catalyst surface. A *cell* is represented by a pair $(u, (i,j))$, where $u \in A$ is a cell state, $(i,j) \in X$ is a cell name. On the $X$ *naming functions* $\varphi(i,j) : X \to X$, determining one of the neighbours for cell $(i,j)$, are introduced. A set of naming functions determines an *underlying template* $T(i,j)$, defining nearest neighbours of cell $(i,j)$ [1]. In the model under investigation the following templates (Fig. 1) are used.

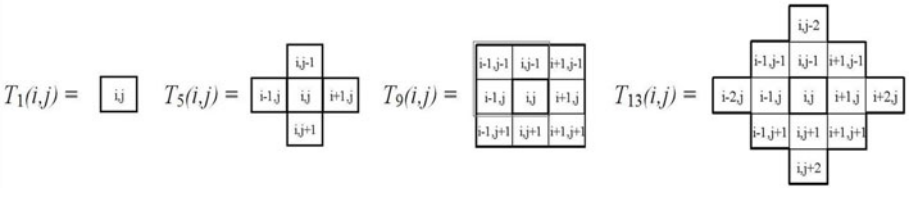



**Fig. 1.** Underlying templates used in CA-model of oxidation reaction

Local operator $\Theta(i,j)$ is a complex composition of substitutions $\theta_l, l \in \{2,3,4,5\}$ and their superpositions $\theta_{(l,9)} = \theta_9(\theta_l),\ l \in \{1,6,7,8\}$: $\Theta(i,j) = \{\theta_{(1,9)}, \theta_2, \theta_3, \theta_4, \theta_5, \theta_{(6,9)}, \theta_{(7,9)}, \theta_{(8,9)}\}$. Each substitution $\theta_l \in \Theta(i,j)$ corresponds to the elementary stage: $\theta_1$ is CO adsorptopn, $\theta_2$ and $\theta_3$ are CO desorption on $*_{\text{hex}}$ and $*_{1\times1}$, $\theta_4$ and

$\theta_5$ are phase transitions hex $\rightarrow 1\times 1$ and $1\times 1 \rightarrow$ hex, $\theta_6$ and $\theta_7$ are $O_2$ adsorption on $*_{\text{hex}}$ and $*_{1\times 1}$, $\theta_8$ is $CO_{\text{ads}}$ diffusion, $\theta_9$ is interaction between $CO_{\text{ads}}$ and $O_{\text{ads}}$.

Application of $\Theta(i,j)$ to cell $(i,j)$ consists of a choice of one of $\theta_l$ or $\theta_{(l,9)}, l = \{1,...,8\}$ with probability $p_l$, and of calculation of new cells states in $T(i,j)$. Probabilities $p_l$ are calculated according to the rate coefficients $k_l$ given in [4]. The $\theta_9$ is realized immediately after $\theta_l, l \in \{1,6,7,8\}$. Therefore, superposition $\theta_{(l,9)}$ is used.

Application of $\theta_{(l,9)} = \theta_9(\theta_l),\ l = 6,7,8$, requires to use the template $T_{13}(i,j)$, which is the union of $\theta_l$ and $\theta_9$ templates: $T_{13}(i,j) = \bigcup_{k=1}^{4} T_5(\varphi_k(i,j))$. Superposition $\theta_{(8,9)}$ applies $M_{\text{diff}}$ times more frequently than other substitutions, parameter of CO diffusion intensity $M_{\text{diff}} = 50$ [4].

The asynchronous mode of CA prescribes the local operator $\Theta(i,j)$ to be applied to a randomly chosen cell $(i,j) \in X$, changing its state immediately. The simulation process is divided into iterations, an iteration being $\mid X \mid \cdot M_{\text{diff}}$ application of substitutions $\theta_l \in \Theta(i,j)$ to randomly chosen cells. An iteration transfers the cellular array $\Omega(t)$ into $\Omega(t+1)$, where $t$ is iteration number. Sequence $\sum(\Omega) = \Omega(0), \ldots, \Omega(t), \Omega(t+1), \ldots, \Omega(t_{\text{fin}})$ is named evolution.

### 2.2 Results of ACA Sequential Implementation Simulating CO Oxidation Reaction

The CO oxidation reaction has been simulated by sequential ACA implementation with cellular array size $\mid X \mid = 200\times 200$ cells and periodic boundary conditions. At the initial moment all cells states are $*_{\text{hex}}$.

In the course of the simulation the following characteristics are obtained.

1. Coverages of oxygen adsorbed on the platinum surface $n(O_{\text{ads}})$, obtained after each iteration as a ratio of the number of cells in states $O_{\text{ads}}^{1\times 1}$ and $O_{\text{ads}}^{\text{hex}}$ to the cellular array size $(M_i \cdot M_j)$.
2. Coverages of adsorbed CO $n(CO_{\text{ads}})$, obtained by analogy $n(O_{\text{ads}})$.
3. Portions of surface with square and hexagonal structure: $n(1\times 1), n(\text{hex})$, calculated after each iteration.
4. Intensity of $CO_2$ formation: $v(CO_2)$, computed as the number of interactions between $CO_{\text{ads}}$ and $O_{\text{ads}}$ per iteration divided by $M_i \cdot M_j$.

As a result of ACA simulation the oscillating behavior of the reagents coverage and $n(1\times 1), n(\text{hex})$ are observed which are in agreement with experimental data (Fig. 2).

Oscillations in the reaction are observed only at specified values of rate coefficients. To determine the domain of oscillations existence the bifurcation diagram in the phase space of $k_1$ and $k_6$ (Fig. 3) has been obtained by performing more than 100 numerical experiments for $k_1 \in [0;200]$ and $k_6 \in [0;10^5]$.

As a result, four reaction behavior patterns are found: two equilibrium states - below curve 1 and above curve 3, and two oscillatory regime - area between curves 1 and 2 and area between curves 2 and 3.

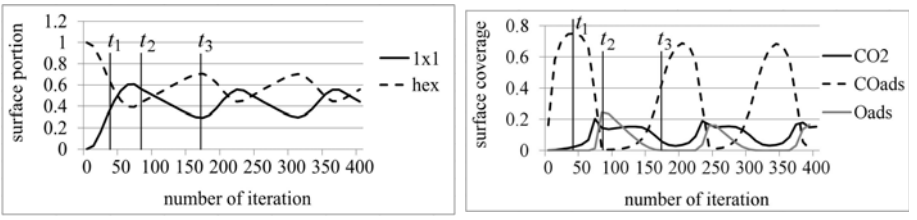



**Fig. 2.** Oscillatory character of CO oxidation reaction over Pt: a) portions of surface with $1 \times 1$ and hex structure; b) surface coverage of $CO_{ads}$, $O_{ads}$ and intensity of $CO_2$ formation

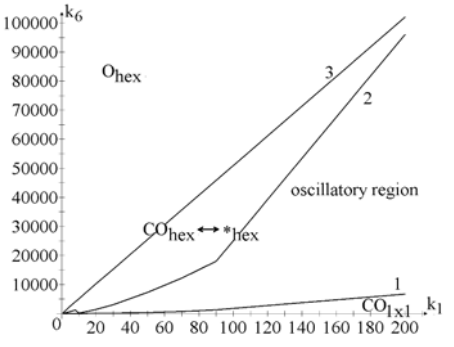



**Fig. 3.** The bifurcation diagram of CO oxidation reaction over Pt

The bifurcation diagram represents possible states of dynamic system and can be used as characteristic of the whole of system behavior.

## 3 Parallel Implementation of ACA-Model Simulating the CO Oxidation Reaction over Pt(100)

### 3.1 Approximation of ACA by a Block-Synchronous CA

The direct parallelization of ACA is not efficient, therefore the transformation of ACA to BSCA is to be used [2, 3]. BSCA $\aleph_\beta = \langle A, X, \Theta \rangle$ is constructed as follows.

1. On the naming set $X$ a template named the block $B(i,j)$ is defined, which should include the underlying templates (Fig. 1):

$$T_1(i,j) \subset T_5(i,j) \subset T_9(i,j) \subset T_{13}(i,j) \subseteq B(i,j) \Rightarrow \quad B(i,j) = T_{13}(i,j). \qquad (1)$$

The block $B(i,j)$ defines on $X$ a set of partitions $\Pi = \{X_1, X_2, \ldots X_{13}\}$:

$$| X_k | = \frac{| X |}{13}, \quad \bigcup_{k=1}^{13} X_k = X, \quad X_k \bigcap X_l = \emptyset, \quad k,l \in \{1,\ldots,13\}; \qquad (2)$$

$$\bigcup_{(i,j)\in X_k} B(i,j) = X, \quad B(i,j) \bigcap B(g,h) = \emptyset, \quad (i,j),(g,h) \in X_k. \qquad (3)$$

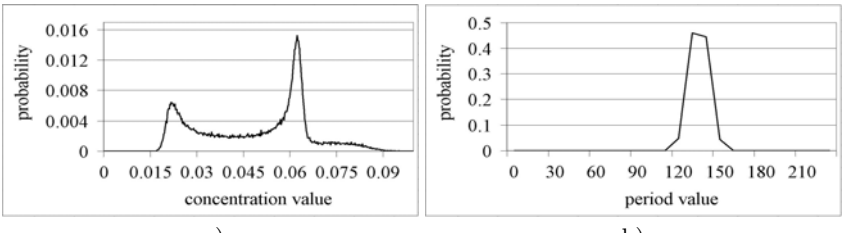
2. An iteration is divided into 13 steps. On each step the local operator $\Theta(i,j)$ is applied synchronously to cells of randomly chosen partition $X_k, k = 1,\ldots,13$. The choice order of cells of $X_k$ is unimportant, because (1) ensures that application $\Theta(i,j)$ to the cells from different blocks are independent.

### 3.2 ACA and BSCA Evolutions Comparison

To assess the accuracy of ACA approximation by BSCA simulation the following characteristics have been obtained by computer experiments:

1. probability distribution of $n(\mathrm{O_{ads}}), n(\mathrm{CO_{ads}}), v(\mathrm{CO_2}), n(1\times 1), n(\mathrm{hex})$ and of oscillation periods $T$;
2. mathematical expectation and dispersion of $n(\mathrm{O_{ads}})$, $n(\mathrm{CO_{ads}})$, $v(\mathrm{CO_2})$, $n(1\times 1)$, $n(\mathrm{hex})$, $T$ and the confidence intervals of its;
3. bifurcation diagram of oxidation reaction.

Probability distribution of the random quantities $n(\mathrm{O_{ads}})$, $n(\mathrm{CO_{ads}})$, $v(\mathrm{CO_2})$, $n(1\times 1)$, $n(\mathrm{hex})$ and $T$ have been computed for the sample size $t_{\mathrm{fin}} = 10^6$ iterations. For example, in Fig.4 the probability distribution of $v(\mathrm{CO_2})$ and $T$ are shown. Difference between the probability distribution of characteristics




a) b)

**Fig. 4.** The probability distribution of a) intensity of $\mathrm{CO_2}$ formation and b) oscillation periods

obtained by ACA and BSCA are quite admissible. Root-mean-square differences of the probability distributions of $v(\mathrm{CO_2})$ and $T$ are equal $E(v(\mathrm{CO_2})) = 5.9\cdot 10^{-5}$ and $E(T) = 7.1\cdot 10^{-5}$.

The mathematical expectation $M\xi$, dispersion $D\xi$ and its confidence intervals $I_{M\xi}$ and $I_{D\xi}$ are computed under the standard formulas with the confidence probability of intervals $\gamma = 0.95$. Table 1 shows statistics for $v(\mathrm{CO_2})$ and $T$ obtained both by ACA and BSCA, its values practically don't differ.

**Table 1.** The statistics values of $v(\mathrm{CO_2})$ and $T$ obtained by ACA and BSCA

| | $M\xi$ | $D\xi$ | $I_{M\xi}$ | $I_{D\xi}$ |
|---|---|---|---|---|
| $v(\mathrm{CO_2})^{ACA}$ | 0.048841 | 0.000317 | (0.048805; 0.048876) | (0.000316; 0.000318) |
| $v(\mathrm{CO_2})^{BSCA}$ | 0.048826 | 0.000316 | (0.048791; 0.048862) | (0.000315; 0.000317) |
| $T^{ACA}$ | 14.490580 | 0.461572 | (14.485427;14.495732) | (0.456621;0.466523) |
| $T^{BSCA}$ | 14.469609 | 0.538590 | (14.464047; 14.475171) | (0.532817; 0.544363) |

Moreover, bifurcation diagrams obtained by ACA and BSCA completely coincide. As follows from above, ACA and BSCA demonstrate the identical behavior pattern of CO oxidation reaction over Pt(100) and approximate ACA by BSCA can be use.

### 3.3 Results of BSCA Parallel Implementation

BSCA parallel implementation on $n$ processors of multicomputer is as follows. The cellular array $\Omega(A, X)$ is divided into $n$ domains $|Dom| = \frac{|X|}{n}$, each being allocated and processed in its own processor.

The computations have been performed on $MVS-100K$ of JSCC RAS. Table 2 shows the efficiency $Q(n) = \frac{T_1}{T_n \cdot n}$ of BSCA parallelization with cellular array size $|X| = 16000 \times 16000$:

**Table 2.** Parallel implementation characteristics of BSCA

| $n$ | 1 | 4 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|---|
| $Q(n)$ | 1 | 0.99 | 0.97 | 0.92 | 0.82 | 0.72 | 0.71 |
| $\|Dom\|$ | $2.56 \cdot 10^8$ | $6.4 \cdot 10^7$ | $1.6 \cdot 10^7$ | $8 \cdot 10^6$ | $4 \cdot 10^6$ | $2 \cdot 10^6$ | $10^6$ |

Table 2 shows that efficiency $Q(n)$ exceeds 80% for 64 processors, decreasing with further increasing of processors number. To achieve high efficiency of BSCA parallel implementation the domain size should be larger than $2.04 \cdot 10^6$ cells.

## 4 Conclusion

Parallel implementation of ACA simulating CO oxidation reaction over Pt(100) surface is proposed. Parallel implementation of ACA is performed by approximating it by BSCA. Approximation accuracy is investigated by the comparative analysis of ACA and BSCA simulation results. The analysis has shown that the differences of probability distributions, mathematical expectation and dispersion for ACA and BSCA is less than $10^{-4}$, and bifurcation diagrams of the reaction completely coincide. The efficiency of BSCA parallel implementation exceeding 80% for domain size larger than $2.04 \cdot 10^6$ cells is obtained.

## References

[1] Bandman, O.L.: Cellular Automatic Models of Spatial Dynamics. System informatics - Methods and models of modern programming 10, 59–113 (2006)
[2] Bandman, O.L.: Parallel simulation of asynchronous cellular automata evolution. In: El Yacoubi, S., Chopard, B., Bandini, S. (eds.) ACRI 2006. LNCS, vol. 4173, pp. 41–47. Springer, Heidelberg (2006)
[3] Nedea, S.V., Lukkien, J.J., Hilbers, P.A.J., Jansen, A.P.J.: Methods for parallel simulations of surface reactions. Advances in computation: Theory and practice, Parallel and distributed scientific and engineering computing 15, 85–97 (2004)
[4] Matveev, A.V., Latkin, E.I., Elokhin, V.I., Gorodetskii, V.V.: Monte Carlo model of oscillatory CO oxidation having regard to the change of catalytic properties due to the adsorbate-induced Pt(100) structural transformation. Journal of Molecular Catalysis A: Chemical 166, 23–30 (2001)

# Smallest Implementations of Optimum-Time Firing Squad Synchronization Algorithms for One-Bit-Communication Cellular Automata

Hiroshi Umeo* and Takashi Yanagihara

Univ. of Osaka Electro-Communication,
Neyagawa-shi, Hastu-cho, 18-8, Osaka, 572-8530, Japan
umeo@cyt.osakac.ac.jp

**Abstract.** Synchronization of large-scale networks is an important and fundamental computing primitive in parallel and distributed systems. The firing squad synchronization problem (FSSP) on cellular automata (CA) has been studied extensively for more than fifty years, and a rich variety of synchronization algorithms has been proposed for not only one-dimensional but two-dimensional arrays. In the present paper, we study the FSSP on 1-bit-communication cellular automata, $CA_{1-bit}$. The $CA_{1-bit}$ is a weakest subclass of CAs in which the amount of inter-cell communication bits transferred among neighboring cells at one step is restricted to 1-bit. We propose two state-efficient implementations of optimum-time FSSP algorithms for the $CA_{1-bit}$ and show that the communication restriction has no influence on the design of optimum-time FSSP algorithms. The implementations proposed are the smallest ones, known at present.

## 1 Introduction

Synchronization of large-scale networks is an important and fundamental computing primitive in parallel and distributed systems. We study a synchronization problem that gives a finite-state protocol for synchronizing cellular automata. The synchronization in cellular automata has been known as the firing squad synchronization problem (FSSP) since its development, in which it was originally proposed by J. Myhill in the book edited by Moore [1964] to synchronize all/some parts of self-reproducing cellular automata. The problem has been studied extensively for more than fifty years [1-25].

In the present paper, we study the FSSP on 1-bit-communication cellular automata, $CA_{1-bit}$. The $CA_{1-bit}$ is a weakest subclass of CAs in which the amount of inter-cell communication bits transferred among neighboring cells at one step is restricted to 1-bit. We propose several state-efficient implementations of optimum- and non-optimum-time FSSP algorithms for the one- and two-dimensional $CA_{1-bit}$. The implementations proposed are the smallest ones, known at present.

Specifically, we attempt to answer the following questions:

* Corresponding author.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 210–223, 2011.

- Can the $CA_{1-bit}$ provide time-efficient solutions to problems solved by conventional (i.e. constant-bit transfer communication) cellular automata without any time overhead?
- What is the smallest transition rule set in the 1-bit realizations of FSSP algorithms?
- What is an exact transition rule set for the optimum-time 2-D square array synchronization on $CA_{1-bit}$?

In Section 2 we introduce a class of the $CA_{1-bit}$. Section 3 presents two 1-bit implementations of an optimum- and non-optimum-time FSSP algorithm on the one-dimensional $CA_{1-bit}$. In Section 4 an implementation of an optimum-time square synchronization algorithm on the $CA_{1-bit}$ is presented.

## 2 One-Bit-Communication Cellular Automata

A one-dimensional 1-bit inter-cell communication cellular automaton (abbreviated as 1-D $CA_{1-bit}$ ) consists of a finite array of identical finite state automata, each located at a positive integer point. See Fig. 1. Each automaton is referred to as a cell. The cell at point $i$ is denoted by $C_i, 1 \leq i \leq n$, where $n$ is any positive integer such that $n \geq 2$. Each $C_i$, except for $C_1$ and $C_n$, is connected with its left and right neighbor cells via a left or right one-way communication link, where those communication links are indicated by right- and left-going arrows, respectively, shown in Fig. 1. Each one-way communication link can transmit only one bit at each step in each direction.

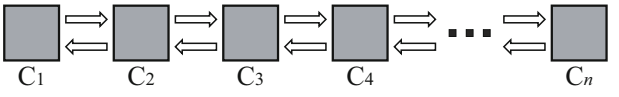



**Fig. 1.** One-dimensional cellular automaton connected with 1-bit inter-cell communication links

A cellular automaton with 1-bit inter-cell communication, $CA_{1-bit}$, consists of a finite array of a finite state automaton $A = (Q, \delta)$, where

1. $Q$ is a finite set of internal states.
2. $\delta$ is a function that defines the next state of any cell and its binary outputs to its left and right neighbor cells such that $\delta$: $Q \times \{0,1\} \times \{0,1\} \rightarrow Q \times \{0,1\} \times \{0,1\}$, where $\delta(p,x,y) = (q,x',y')$, $p, q \in Q$, $x, x', y, y' \in \{0,1\}$, has the following meaning: We assume that, at step $t$, the cell $C_i$ is in state $p$ and receives binary inputs $x$ and $y$ from its left and right communication links, respectively. Then, at the next step $t+1$, $C_i$ takes a state $q$ and outputs $x'$ and $y'$ to its left and right communication links, respectively. Note that binary inputs to $C_i$ at step $t$ are also outputs of $C_{i-1}$ and $C_{i+1}$ at step $t$. A quiescent state $q \in Q$ has a property such that $\delta(q,0,0) = (q,0,0)$.

The $\mathrm{CA}_{1-\mathrm{bit}}$ is a special subclass of *normal* (i.e., O(1)-bit communication) cellular automata. The O(1)-bit communication model is a *conventional* CA in which the amount of communication bits exchanged in one step between neighboring cells is assumed to be O(1) bits. However, such bit amounts exchanged between inter-cells are hidden behind the definition of conventional automata-theoretic finite state descriptions. On the other hand, the 1-bit inter-cell communication model studied in the present paper is a new subclass of conventional CAs, in which inter-cell communication is restricted to 1-bit communication. The number of internal states of the $\mathrm{CA}_{1-\mathrm{bit}}$ is assumed to be finite as in a usual sense. The next state of each cell is determined based on the present state of the cell and two binary 1-bit inputs from its left and right neighbor cells. Thus, the $\mathrm{CA}_{1-\mathrm{bit}}$ is one of the weakest and simplest models among the variants of the conventional CAs.

Let $N$ be any normal cellular automaton with a set of states $Q$ and a transition function $\delta : Q^3 \to Q$. The state of each cell on $N$ depends on the cell's previous state and states on its nearest neighbor cells. This means that the total information exchanged per step between neighboring cells is O(1) bits. Each state in $Q$ can be encoded with a binary sequence of length $\lceil \log_2 |Q| \rceil$ and then sending the binary sequences sequentially bit-by-bit in each direction via each one-way communication link. Those sequences are then received bit-by-bit and decoded into their corresponding states in $Q$. It is easily seen that these encoding and decoding procedures can be described in terms of finite states. Thus, the $\mathrm{CA}_{1-\mathrm{bit}}$ can simulate one step of $N$ in $\lceil \log_2 |Q| \rceil$ steps. This observation gives the following computational relation between the normal CA and the $\mathrm{CA}_{1-\mathrm{bit}}$.

**Theorem 1.** Mazoyer [1996], Umeo and Kamikawa [2001] Let $N$ be any *normal* cellular automaton operating in $T(n)$ steps with internal state set $Q$. Then, there exists a $\mathrm{CA}_{1-\mathrm{bit}}$ that can simulate $N$ in $kT(n)$ steps, where $k$ is a positive constant integer such that $k = \lceil \log_2 |Q| \rceil$.

A question is whether the $\mathrm{CA}_{1-\mathrm{bit}}$ can solve many variants of the FSSP problems solved by conventional cellular automata without any overhead in time complexities. We answer the question by presenting several optimum-time solutions.

## 3 Firing Squad Synchronization Problem

Section 3 studies the FSSP on the 1-D $\mathrm{CA}_{1-\mathrm{bit}}$, the solution of which yields a finite-state 1-bit-communication protocol for synchronizing 1-D $\mathrm{CA}_{1-\mathrm{bit}}$. A rich variety of synchronization algorithms have been proposed on the O(1)-bit communication models. An optimum-time (i.e., $(2n-2)$-step) synchronization algorithm for one-dimensional array of length $n$ was devised first by Goto [1962]. The algorithm needed many thousands of internal states for its realization. Afterwards, Waksman [1966], Balzer [1967], Gerken [1987] and Mazoyer [1987] developed an optimum-time algorithm and reduced the number of states realizing the algorithm, each with 16, 8, 7 and 6 states on the conventional O(1)-bit communication model.

The FSSP on the 1-D $CA_{1-bit}$ is defined as follows: At time $t = 0$, the left end cell $C_1$ is in the *fire-when-ready* state, which is an initiation signal for the array. The FSSP is to determine a description (a state set $\mathcal{Q}$ and a next-state function such that $\delta$: $\mathcal{Q} \times \{0,1\} \times \{0,1\} \rightarrow \mathcal{Q} \times \{0,1\} \times \{0,1\}$) for cells that ensures all cells enter the *fire* state at exactly the same time and for the first time. The set of internal states and the next-state function must be independent of $n$. We sometimes use a word "rule" as a value of the next-state function of a given solution.

### 3.1 An Optimum-Time FSSP Algorithm on 1-D $CA_{1-bit}$

Here we briefly sketch the design scheme for the firing squad synchronization algorithm. Figure 2 (left) is a space-time diagram for the Waksman-Balzer-Gerken-type optimum-step firing squad synchronization algorithm. The general at time $t = 0$ emits an infinite number of signals which propagate at $1/(2^k - 1)$ speed, where $k$ is any positive integer such that $k \geq 1$. These signals meet with a

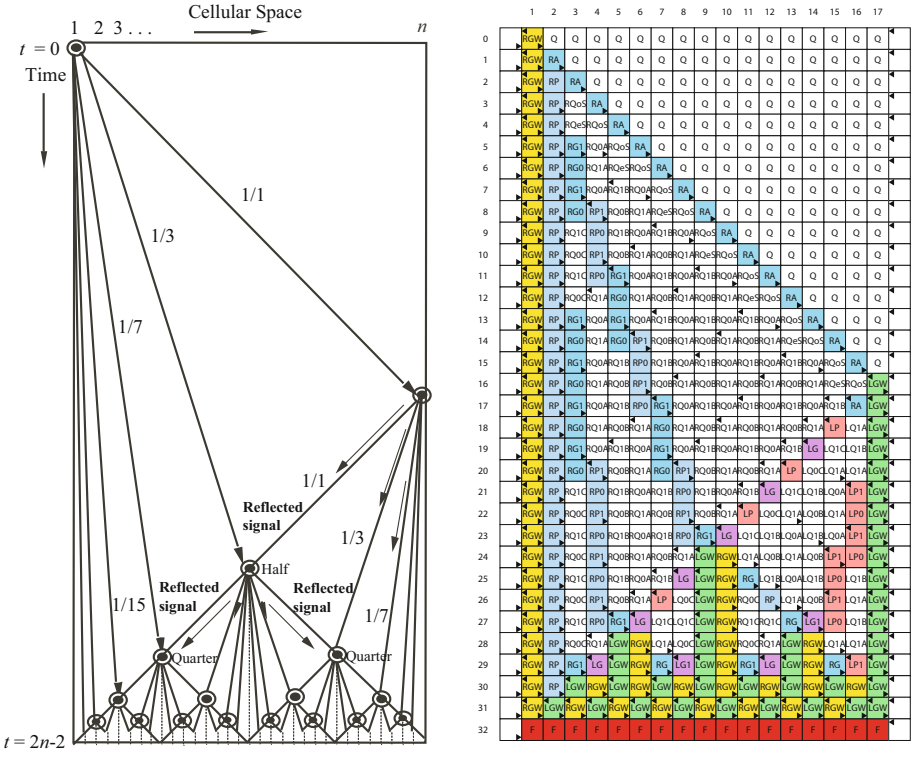



**Fig. 2.** Space-time diagram for optimum-time synchronization algorithms with a general at left end (left) and some snapshots of the synchronization processes for the 35-state implementation on $CA_{1-bit}$ (right)

reflected signal at half point, quarter points, ..., etc., denoted by $\odot$ in Fig. 2 (left). It is noted that those cells indicated by $\odot$ are synchronized at each corresponding step. By increasing the number of synchronized cells exponentially, eventually all of the cells are synchronized exactly at time $t = 2n - 2$. It is the time when the first reflected signal arrives at the left end.

Most of the implementations for the optimum-time synchronization algorithms on $\mathrm{CA}_{1-\mathrm{bit}}$ developed so far are based on the diagram shown in Fig. 2 (left). Mazoyer [1996] developed an optimum-time synchronization algorithm for the $\mathrm{CA}_{1-\mathrm{bit}}$ based on the Balzer's 8-state realization in Balzer [1967]. Each cell of the constructed $\mathrm{CA}_{1-\mathrm{bit}}$ had 58 internal states in Mazoyer [1996]. Nishimura, Sogabe and Umeo [2003] also constructed an optimum-time synchronization algorithm (**NSU** algorithm for short) based on the Waksman's 16-state implementation in Waksman [1966]. Each cell had 78 internal states and 208 transition rules.

**Theorem 2.** Mazoyer [1996], Nishimura, Sogabe and Umeo [2003] There exists a $\mathrm{CA}_{1-\mathrm{bit}}$ that can synchronize $n$ cells with the general at left end of the array in $2n - 2$ steps.

Umeo, Yanagihara and Kanazawa [2006] developed a non-optimum-step synchronization algorithm for the $\mathrm{CA}_{1-\mathrm{bit}}$ based on Mazoyer's 6-state O(1)-bit communication algorithm in Mazoyer [1987]. The constructed $\mathrm{CA}_{1-\mathrm{bit}}$ can synchronize $n$ cell in $2n - 1$ steps, that is, 1 step slower than optimum-time, and each cell has 54 states and 207 transition rules.

**Theorem 3.** Umeo, Yanagihara and Kanazawa [2006] There exists a 54-state $\mathrm{CA}_{1-\mathrm{bit}}$ that can synchronize any $n$ cells in $2n - 1$ non-optimum-step.

Here we construct a smaller optimum-time implementation based on Gerken's 7-state O(1)-bit communication synchronization algorithm presented in Gerken [1987]. The constructed $\mathrm{CA}_{1-\mathrm{bit}}$ has 35 internal states and 114 transition rules. The set of 35 states is {`Q, RGW, RPW, RA, RQoS, RQeS, RQ1A, RQ0A, RQ1B, RQ0B, RQ1C, RQ0C, RG1, RG0, RP1, RP0, RG, RP, LGW, LPW, LQ1A, LQ1B, LQ0A, LQ0B, LQ1C, LQ0C, LG1, LG0, LP1, LP0,LG, LP, LP', QW, F`}, where the state `Q` is the quiescent state, `RGW` is the general state, and `F` is the firing state, respectively. Table 1 presents the transition rule set for the 35-state synchronization protocol. Note that state transition from the firing state `F` is not included in the Table 1, since it is the final state. Figure 2 (right) shows some snapshots for synchronization processes on 17 cells on the $\mathrm{CA}_{1-\mathrm{bit}}$. The small right- and left-facing black triangles, ▶ and ◀ in the figure, indicate a 1-bit signal transfer in the right or left direction between neighbor cells. The symbol in each cell shows its internal state.

**Theorem 4.** There exists a 35-state $\mathrm{CA}_{1-\mathrm{bit}}$ that can synchronize $n$ cells in $2n - 2$ optimum-steps.

**Table 1.** Transition table for a 35-state implementation of the optimum-time synchronization algorithm

| 1 Q | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (Q,0,0) | (Q,0,0) |
| L = 1 | (RA,0,1) | (LGW,1,1) |

| 2 RGW | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RGW,0,1) | (F,0,0) |
| L = 1 | (RGW,1,1) | (F,0,0) |

| 3 RPW | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RPW,0,1) | (LGW,1,1) |
| L = 1 | -- | -- |

| 4 RA | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQoS,0,0) | -- |
| L = 1 | (RP,0,0) | -- |

| 5 RQoS | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ0A,0,1) | (LP',1,0) |
| L = 1 | (RQeS,0,0) | (LP,1,0) |

| 6 RQeS | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ1B,1,0) | -- |
| L = 1 | (RG1,0,1) | -- |

| 7 RQ1A | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ0A,0,0) | (LG,1,0) |
| L = 1 | -- | -- |

| 8 RQ0A | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ1A,0,0) | (RQ1A,1,0) |
| L = 1 | (RQ1A,0,0) | (RP1,1,1) |

| 9 RQ1B | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ0B,0,0) | (LP,1,0) |
| L = 1 | -- | -- |

| 10 RQ0B | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ1B,0,0) | (RQ1B,1,0) |
| L = 1 | (RQ1B,0,0) | (RG1,1,1) |

| 11 RQ1C | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LQ1B,0,0) | (LQ1C,0,0) |
| L = 1 | (RQ0C,0,0) | (LP,1,0) |

| 12 RQ0C | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LQ1A,0,1) | -- |
| L = 1 | (RQ1C,0,0) | (RG1,0,1) |

| 13 RG1 | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RG0,0,0) | (LPW,1,0) |
| L = 1 | (RG0,0,0) | (LPW,1,0) |

| 14 RG0 | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RG1,0,1) | (RQ1B,0,0) |
| L = 1 | (RG1,0,1) | (RQ1C,0,0) |

| 15 RP1 | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RP0,0,0) | (LGW,1,1) |
| L = 1 | -- | -- |

| 16 RP0 | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RP1,0,1) | (RQ1A,1,0) |
| L = 1 | -- | -- |

| 17 RG | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LQ1C,0,0) | (LPW,1,0) |
| L = 1 | (LQ1C,0,0) | (LPW,1,0) |

| 18 RP | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LQ0C,0,0) | (LGW,1,1) |
| L = 1 | (RP,0,1) | (LGW,1,1) |

| 19 LGW | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LGW,1,0) | (LGW,1,1) |
| L = 1 | (F,0,0) | (F,0,0) |

| 20 LPW | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LPW,1,0) | -- |
| L = 1 | (RGW,1,1) | -- |

| 21 LQ1A | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LQ0A,0,0) | (LQ1B,0,0) |
| L = 1 | (RG,0,1) | (LP1,1,0) |

| 22 LQ1B | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LQ0B,0,0) | (LQ1A,0,0) |
| L = 1 | (RP,0,1) | (RP,0,0) |

| 23 LQ0A | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LQ1A,0,0) | (LQ1A,0,0) |
| L = 1 | (LQ1A,0,1) | (LP1,1,1) |

| 24 LQ0B | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LQ1B,0,0) | (LQ1B,0,0) |
| L = 1 | (LQ1B,0,1) | (LG1,1,1) |

| 25 LQ1C | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ1B,0,0) | (LQ0C,0,0) |
| L = 1 | (RQ1C,0,0) | (RP,0,1) |

| 26 LQ0C | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ1A,1,0) | (LQ1C,0,0) |
| L = 1 | -- | (LG1,1,0) |

| 27 LG1 | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LG0,0,0) | (LG0,0,0) |
| L = 1 | (RPW,0,1) | (RPW,0,1) |

| 28 LG0 | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LG1,1,0) | (LG1,1,0) |
| L = 1 | (LQ1B,0,0) | (LQ1C,0,0) |

| 29 LP1 | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LP0,0,0) | (LP0,0,0) |
| L = 1 | (RGW,1,1) | (RPW,0,0) |

| 30 LP0 | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (LP1,1,0) | (LP1,1,0) |
| L = 1 | (LQ1A,0,1) | (LQ1B,0,0) |

| 31 LG | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ1C,0,0) | (RQ1C,0,0) |
| L = 1 | (RPW,0,1) | (RPW,0,1) |

| 32 LP | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (RQ0C,0,0) | (LP,1,0) |
| L = 1 | (RGW,1,1) | (RGW,1,1) |

| 33 LP' | R = 0 | R = 1 |
|---|---|---|
| L = 0 | -- | (LQ1A,0,0) |
| L = 1 | -- | (RPW,0,0) |

| 34 QW | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (QW,0,0) | -- |
| L = 1 | (LGW,1,0) | -- |

### 3.2 A $3n$-Step Non-optimum-Time FSSP Algorithm on 1-D $\mathbf{CA_{1-bit}}$

The $3n$-step algorithm that synchronizes $n$ cells in $3n \pm O(\log n) + O(1)$ steps is an interesting class of synchronization algorithms due to its simplicity and straightforwardness and it is important in its own right in the design of generic cellular algorithms. Minsky and MacCarthy [1967] gave an idea for designing the $3n$-step synchronization algorithm and Fischer [1965] implemented the $3n$-step algorithm, yielding a 15-state implementation. Yunès [1994] proposed a seven-state $3n$-step firing squad synchronization algorithm. Umeo et al. [2006] and Yunès [2008] proposed six-state $3n$-step symmetrical synchronization algorithms, both known as the smallest $3n$-step FSSP solutions for the O(1)-bit communication model.

Figure 3 (left) shows a space-time diagram for the well-known $3n$-step firing squad synchronization algorithm. The synchronization process can be viewed

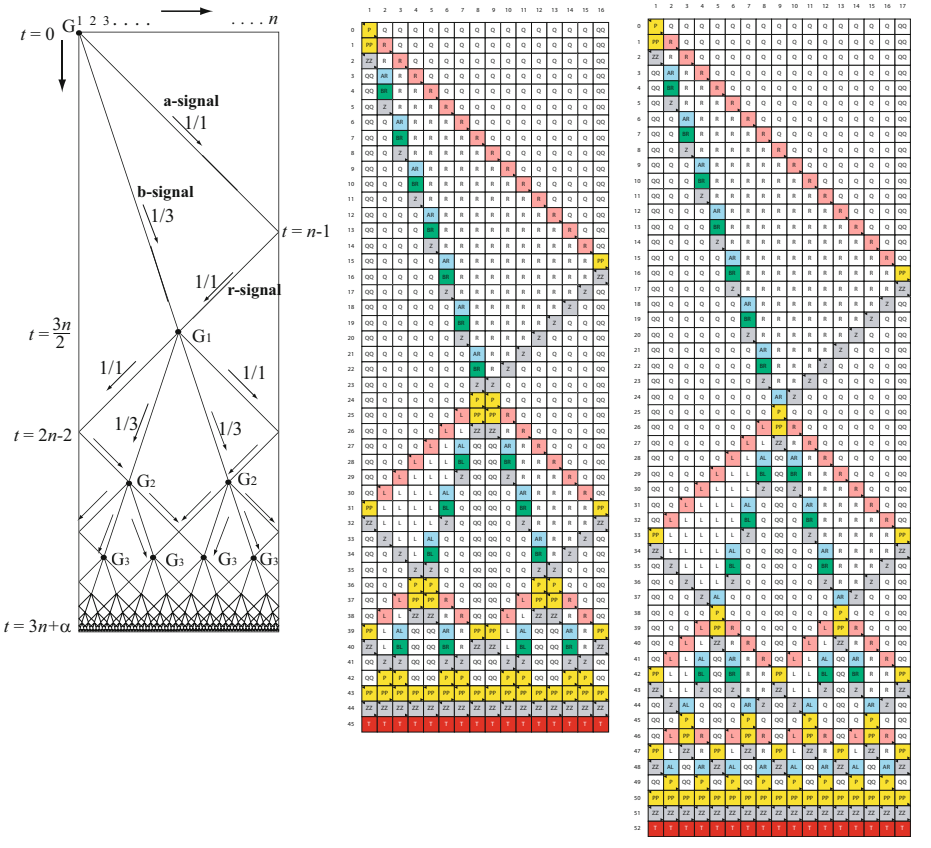



**Fig. 3.** A space-time diagram for the $3n$-step FSSP algorithm (left) and snapshots for synchronization processes of the 13-state implementation of the algorithm on 16 and 17 cells (middle and right)

as a typical divide-and-conquer strategy that operates in parallel in the cellular space. An initial *general* G, located at left end of the array of length $n$, generates simultaneously two special signals, referred to as *a-signal* and *b-signal*, which propagate in the right direction at a speed of 1/1 (i.e., 1 cell per unit step) and 1/3, respectively. The a-signal arrives at the right end at time $t = n - 1$, reflects there immediately, then continues to move at the same speed in the left direction. The reflected signal is referred to as *r-signal*. The b- and r-signals meet at one or two center cells, depending on the parity of $n$. In the case that $n$ is odd, the cell $C_{\lceil n/2 \rceil}$ becomes a *General* at time $t = 3\lceil n/2 \rceil - 2$. The new *general* works for synchronizing both its left and right halves of the cellular space divided. Note that the *general* is shared by the two halves. In the case that $n$ is even, two cells $C_{\lceil n/2 \rceil}$ and $C_{\lceil n/2 \rceil + 1}$ become the next *general* at time $t = 3\lceil n/2 \rceil$. Each *general* works for synchronizing its left and right halves of the cellular space, respectively.

Thus at time

$$t = \begin{cases} 3\lceil n/2 \rceil - 2 & n\text{: odd} \\ 3\lceil n/2 \rceil & n\text{: even,} \end{cases} \quad (1)$$

the array knows its center point(s) and generates one or two new *general*($s$) $G_1$. The new *general*($s$) $G_1$ generates the same 1/1- and 1/3-speed signals in both left and right directions and repeat the same procedures as above. Thus, the original synchronization problem of size $n$ is divided into two sub-problems of size $\lceil n/2 \rceil$. In this way, the original array is split into equal two, four, eight, ..., subspaces synchronously. In the last, the original problem of size $n$ can be split into small sub-problems of size 2. Most of the $3n$-step synchronization algorithms developed so far are based on the similar scheme.

A one-bit implementation presented in this paper is also based on the synchronization scheme above. Table 2 presents the transition rule set for the 13-state synchronization protocol. The set of 13 states is {`Q, QQ, P, PP, Z, AR, BR, AL, BL, R, L, ZZ, T`}, where the state `Q` is the quiescent state, `P` is the general state, and `T` is the firing state, respectively. The table consists of 32 transition rules.

**Table 2.** Transition table for the 13-state $3n$-step non-optimum-time synchronization algorithm

| 1 Q | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (Q,0,0) | (L,1,0) |
| L = 1 | (R,0,1) | -- |

| 2 QQ | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (QQ,0,0) | (PP,1,0) |
| L = 1 | (PP,0,1) | (PP,0,0) |

| 3 P | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (PP,0,0) | (PP,0,1) |
| L = 1 | (PP,1,0) | -- |

| 4 PP | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (ZZ,1,1) | (ZZ,1,0) |
| L = 1 | (ZZ,0,1) | -- |

| 5 Z | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (Q,0,0) | (P,1,1) |
| L = 1 | (P,1,1) | -- |

| 6 AR | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (BR,0,0) | (P,1,1) |
| L = 1 | -- | -- |

| 7 BR | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (Z,0,1) | -- |
| L = 1 | -- | -- |

| 8 AL | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (BL,0,0) | -- |
| L = 1 | (P,1,1) | -- |

| 9 BL | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (Z,1,0) | -- |
| L = 1 | -- | -- |

| 10 R | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (R,0,0) | (Z,1,0) |
| L = 1 | (AR,0,0) | -- |

| 11 L | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (L,0,0) | (AL,0,0) |
| L = 1 | (Z,0,1) | -- |

| 12 ZZ | R = 0 | R = 1 |
|---|---|---|
| L = 0 | (QQ,0,0) | (T,0,0) |
| L = 1 | (T,0,0) | (T,0,0) |

**Theorem 5.** There exists a 13-state $CA_{1-bit}$ that can synchronize $n$ cells with the general at left end in $3n + O(\log n)$ steps.

Table 3 shows a list of those 1-bit implementations of the FSSP algorithms for $CA_{1-bit}$.

**Table 3.** A list of 1-bit implementation of the FSSP algorithms for $CA_{1-bit}$

| Implementations | # of states | # of rules | Time complexity | Base Algorithms |
|---|---|---|---|---|
| Mazoyer [1996] | 58(61*) | –(167*) | $2n - 2$ | Balzer [1967] |
| Nishimura, Sogabe and Umeo [2003] | 78 | 208 | $2n - 2$ | Waksmann [1966] |
| Umeo, Yanagihara and Kanazawa [2006] | 54 | 207 | $2n - 1$ | Mazoyer [1987] |
| **this paper** | 35 | 114 | $2n - 2$ | Gerken [1987] |
| **this paper** | 13 | 32 | $3n + O(\log n)$ | — |

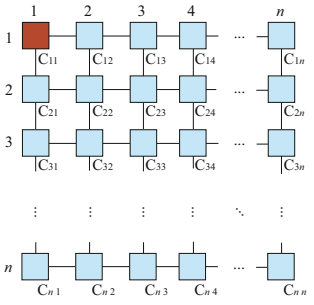


**Fig. 4.** A two-dimensional square cellular automaton

## 4 Firing Squad Synchronization Problem on Two-Dimensional Square Arrays

Figure 4 shows a finite two-dimensional (2-D) square array consisting of $n \times n$ cells, each denoted by $C_{ij}, 1 \leq i, j \leq n$. Each cell is an identical (except the border cells) finite-state automaton. The array operates in lock-step mode in such a way that the next state of each cell (except border cells) is determined by both its own present state and the present states of its north, south, east and west neighbors. Thus, we assume the *von Neumann-type four nearest neighbors*. All cells (*soldiers*), except the north-west corner cell (*general*), are initially in the quiescent state at time $t = 0$ with the property that the next state of a quiescent

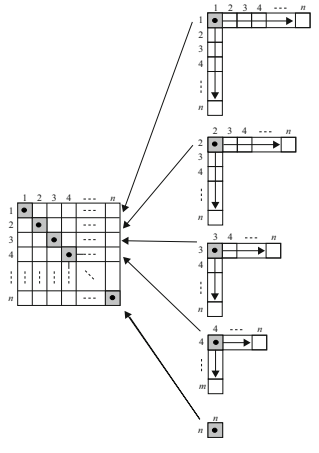


**Fig. 5.** A synchronization scheme for $n \times n$ square cellular automaton. A horizontal and vertical synchronization operations on $L_i$ are mapped onto a square array. A black circle • in a shaded square represents a general on each $L_i$ and a wake-up signal for the synchronization generated by the general is indicated by a horizontal and vertical arrow, respectively.

cell with quiescent neighbors is the quiescent state again. At time $t = 0$, the north-west corner cell $C_{11}$ is in the *fire-when-ready* state, which is the initiation signal for synchronizing the array. The firing squad synchronization problem is to determine a description (state set and next-state function) for cells that ensures all cells enter the *fire* state at exactly the same time and for the first time.

A rich variety of synchronization algorithms for 2-D rectangular cellular automata with O(1)-bit communications has been proposed. Concerning the square synchronization which is a special class of rectangles, several algorithms have been proposed by Beyer [1969], Shinahr [1974], Umeo, Maeda, and Fujiwara [2002], and Umeo and Kubo [2010]. The first optimum-time square synchronization algorithm was proposed by Beyer [1969] and Shinahr [1974]. One can easily see that it takes $2n - 2$ steps for any signal to travel from $C_{11}$ to $C_{nn}$ due to the von Neumann neighborhood. Concerning the time optimality of the two-dimensional square synchronization algorithms, the following theorem has been shown.

**Theorem 6.** Beyer [1969], Shinahr [1974] There exists no cellular automaton that can synchronize any two-dimensional square array of size $n \times n$ in less than $2n - 2$ steps, where the general is located at one corner of the array.

The optimum-time synchronization algorithm proposed by Beyer [1969] and Shinahr [1974] for square arrays operates as follows: We assume that an initial general is located on $C_{11}$. By dividing the entire square array of size $n \times n$ into $n$ *rotated L-shaped* 1-D arrays, shown in Fig. 5, in such a way that the length of the $i$th (from outside) L-shaped array is $2n - 2i + 1$ $(1 \leq i \leq n)$, one treats the square synchronization as $n$ independent 1-D synchronizations with the general located at the bending cell of the L-shaped array. We denote the $i$th L-shaped array by $L_i$ and its horizontal and vertical segment is denoted by $L_i^{\mathrm{h}}$ and $L_i^{\mathrm{v}}$, respectively. Note that a cell at each bending point of the L-shaped array is shared by the two segments. See Fig. 5.

Concerning the synchronization of $L_i$, it can be easily seen that a general is generated at the cell $C_{ii}$ at time $t = 2i - 2$ with the four nearest von-Neumann neighborhood communication, and the general initiates the horizontal (row) and vertical (column) synchronizations on $L_i^{\mathrm{h}}$ and $L_i^{\mathrm{v}}$, each of length $n - i + 1$ via an optimum-time synchronization algorithm which can synchronize arrays of length $\ell$ in $2\ell - 2$ steps. Thus the square array of size $n \times n$ can be synchronized at time $t = 2i - 2 + 2(n - i + 1) - 2 = 2n - 2$ in optimum-steps. In Fig. 5, each general is represented by a black circle $\bullet$ in a shaded square and a wake-up signal for the synchronization generated by the general is indicated by a horizontal and vertical arrow.

The algorithms itself is very simple and now we are going to discuss its implementation in terms of a 2-D cellular automaton. The question is: how many states are required for its realization? Let $Q$ be a set of internal states for the 1-D optimum-time synchronization algorithm which is embedded as a base algorithm. When we implement the algorithm on square arrays based on the scheme above, we usually prepare a different state set used by the cells on $L_i^{\mathrm{h}}$ and $L_i^{\mathrm{v}}$,

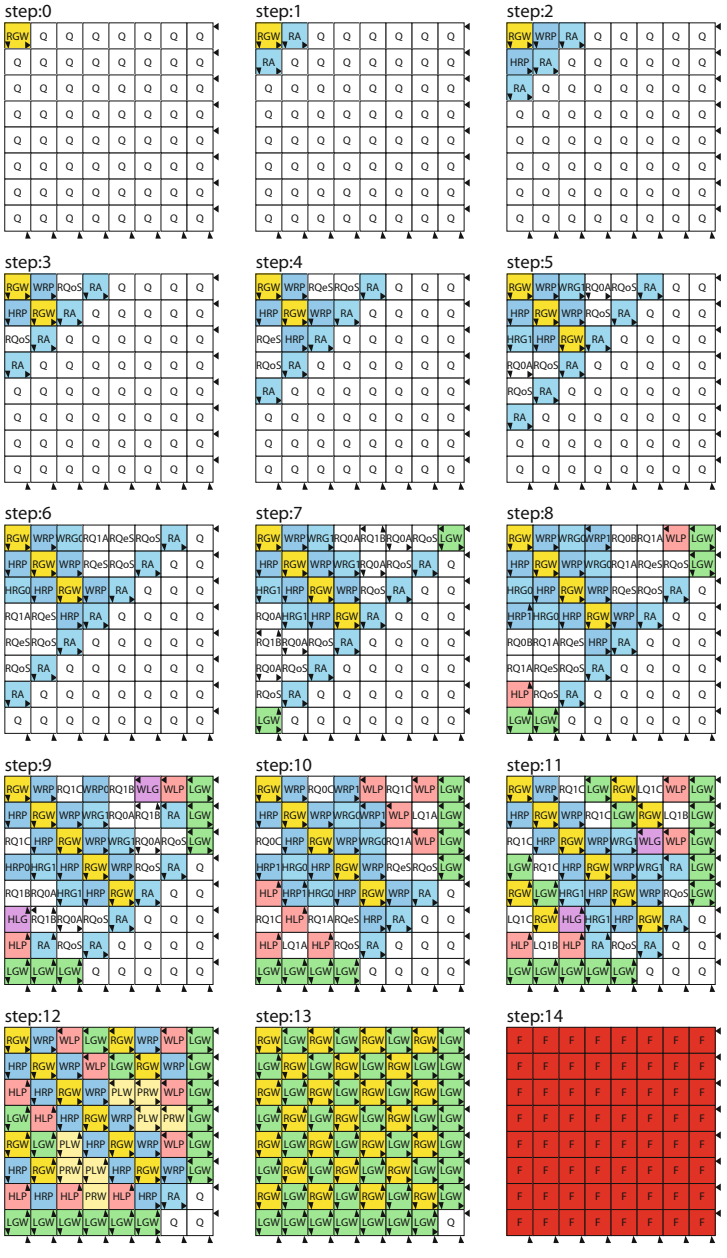



**Fig. 6.** Snapshots for the synchronization processes on a $8 \times 8$ array

**Table 4.** A list of square FSSP algorithms

| Implementations | # of states | # of rules | Time complexity | Communication model | Base algorithms |
|---|---|---|---|---|---|
| Beyer [1969] | — | — | $2n-2$ | O(1)-bit | — |
| Shinahr [1974] | 17 | — | $2n-2$ | O(1)-bit | Balzer [1967] |
| Umeo, Maeda and Fujiwara [2002] | 9 | 1718 | $2n-2$ | O(1)-bit | Mazoyer [1987] |
| Umeo and Kubo [2010] | 7 | 787 | $2n-2$ | O(1)-bit | Mazoyer [1987] |
| Gruska, Torre and Parente [2007] | — | — | $2n-2$ | 1-bit | Mazoyer [1996] |
| **this paper** | 49 | 237 | $2n-2$ | 1-bit | **Theorem 4** (this paper) |

which is in the upper and lower triangle areas separated by the principal diagonal. Thus, $2 \mid\mid Q \mid\mid -1$ states are usually required for its independent row and column synchronization operations in order to avoid state mixing. Only a firing state is shared by the two areas. Shinahr [1974] gave a 17-state implementation based on Balzer's eight-state synchronization algorithm in Balzer [1967]. Later, it has been shown in Umeo, Maeda and Fujiwara [2002] that nine states are sufficient for the optimum-time square synchronization. Recently Umeo and Kubo [2010] improved the square synchronization algorithm by presenting a seven-state time-optimum square synchronizer. Note that those implementations are all for the O(1)-bit communication CAs.

We have implemented the L-shaped algorithm on 2-D $\mathrm{CA}_{1-\mathrm{bit}}$. The number of states and transition rules of the constructed $\mathrm{CA}_{1-\mathrm{bit}}$ is 49 and 237, respectively. Figure 6 shows some snapshots for the synchronization processes on an $8 \times 8$ square array. Due to the space available we omit a complete list of the transition rule set constructed.

**Theorem 7.** There exists a 49-state 2-D $\mathrm{CA}_{1-\mathrm{bit}}$ that can synchronize any $n \times n$ square array in $2n-2$ steps.

In Table 4 we present a list of implementations of the square FSSP algorithms for cellular automata with O(1)-bit and 1-bit communications.

## 5 Conclusions

In the present paper, we have proposed several state-efficient implementations of optimum- and non-optimum-time FSSP algorithms for one- and two-dimensional $\mathrm{CA}_{1-\mathrm{bit}}$. The implementations constructed are the smallest ones in states, known at present.

**Acknowledgements.** The authors would like to express their thanks to reviewers for useful comments. A part of this work is supported by Grant-in-Aid for Scientific Research (C) 21500023.

## References


1. Balzer, R.: An 8-state minimal time solution to the firing squad synchronization problem. Information and Control 10, 22–42 (1967)
2. Beyer, W.T.: Recognition of topological invariants by iterative arrays. Ph.D. Thesis, MIT, p. 144 (1969)
3. Fischer, P.C.: Generation of primes by a one-dimensional real-time iterative array. J. of ACM 12(3), 388–394 (1965)
4. Gerken, H.D.: Über Synchronisations - Probleme bei Zellularautomaten. Diplom arbeit,Institut für Theoretische Informatik, Technische Universität Braunschweig, p. 50 (1987)
5. Goto, E.: A minimal time solution of the firing squad problem. Dittoed course notesfor Applied Mathematics, vol. 298, pp. 52–59. Harvard University, Cambridge (1962)
6. Gruska, J., Torre, S.L., Parente, M.: The firing squad synchronization problem on squares, toruses and rings. Intern. J. of Foundations of Computer Science 18(3), 637–654 (2007)
7. Mazoyer, J.: A six-state minimal time solution to the firing squad synchronization problem. Theoretical Computer Science 50, 183–238 (1987)
8. Mazoyer, J.: On optimal solutions to the firing squad synchronization problem. Theoretical Computer Science 168, 367–404 (1996)
9. Minsky, M.L.: Computation: Finite and infinite machines, pp. 28–29. Prentice-Hall, Englewood Cliffs (1967)
10. Moore, E.F.: The firing squad synchronization problem. In: Moore, E.F. (ed.) Sequential Machines, Selected Papers, pp. 213–214. Addison-Wesley, Reading (1964)
11. Nishimura, J., Sogabe, T., Umeo, H.: A design of o ptimum-time firing squad synchronizationalgorithm on 1-bit cellular automaton. In: Proc. of the 8th International Symposium on Artificial Life and Robotics, pp. 381–386 (2003)
12. Shinahr, I.: Two- and three-dimensional firing squad synchronization problems. Information and Control 24, 163–180 (1974)
13. La Torre, S., Napoli, M., Parente, M.: Firing squad synchronization problem on bidimensional cellular automata with communication constraints. In: Margenstern, M., Rogozhin, Y. (eds.) MCU 2001. LNCS, vol. 2055, pp. 264–275. Springer, Heidelberg (2001)
14. Umeo, H.: Firing squad synchronization problem in cellular automata. In: Meyers, R.A. (ed.) Encyclopedia of Complexity and System Science, vol. 4, pp. 3537–3574. Springer, Heidelberg (2009)
15. Umeo, H.: Problem solving on one-bit-communication cellular automata. In: Hoekstra, A.G., Kroc, J., Sloot, P.M.A. (eds.) Simulating Complex Systems by Cellular Automata, ch. 6, pp. 117–144. Springer, Heidelberg (2010)
16. Umeo, H., Hisaoka, M., Sogabe, T.: A survey on optimum-time firing squad synchronization algorithms for one-dimensional cellular automata. Intern. J. of Unconventional Computing 1, 403–426 (2005)
17. Umeo, H., Maeda, M., Fujiwara, N.: An efficient mapping scheme for embedding any one-dimensional firing squad synchronization algorithm onto two-dimensional arrays. In: Bandini, S., Chopard, B., Tomassini, M. (eds.) ACRI 2002. LNCS, vol. 2493, pp. 69–81. Springer, Heidelberg (2002)
18. Umeo, H., Maeda, M., Hongyo, K.: A design of symmetrical six-state $3n$-step firing squad synchronization algorithms and their implementations. In: El Yacoubi, S., Chopard, B., Bandini, S. (eds.) ACRI 2006. LNCS, vol. 4173, pp. 157–168. Springer, Heidelberg (2006)

19. Umeo, H., Kubo, K.: A seven-state time-optimum square synchronizer. In: Bandini, S., Manzoni, S., Umeo, H., Vizzari, G. (eds.) ACRI 2010. LNCS, vol. 6350, pp. 219–230. Springer, Heidelberg (2010)
20. Umeo, H., Maeda, M., Hisaoka, M., Teraoka, M.: A state -efficient mapping schemefor designing two-dimensional firing squad synchronization algorithms. Fundamenta Informaticae 74(4), 603–623 (2006)
21. Umeo, H., Michisaka, K., Kamikawa, N., Kanazawa, M.: State-efficient one-bit communication solutions for some classical cellular automata problems. Fundamenta Informaticae 78(3), 449–465 (2007)
22. Umeo, H., Yanagihara, T., Kanazawa, M.: State-efficient firing squad synchronization protocols for communication-restricted cellular automata. In: El Yacoubi, S., Chopard, B., Bandini, S. (eds.) ACRI 2006. LNCS, vol. 4173, pp. 169–181. Springer, Heidelberg (2006)
23. Waksman, A.: An optimum solution to the firing squad synchronization problem. Information and Control 9, 66–78 (1966)
24. Yunès, J.B.: Seven-state solution to the firing squad synchronization problem. Theoretical Computer Science 127, 313–332 (1994)
25. Yunès, J.B.: An intrinsically non minimal-time Minsky-like 6 states solution to the firing squad synchronization problem. Theoretical Informatics and Applications 42(1), 55–68 (2008)

# Distributed Genetic Process Mining Using Sampling

Carmen Bratosin, Natalia Sidorova, and Wil van der Aalst

Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{c.c.bratosin,n.sidorova,w.m.p.v.d.aalst}@tue.nl

**Abstract.** Process mining aims at discovering process models from event logs. Complex constructs, noise and infrequent behavior are issues that make process mining a complex problem. A genetic mining algorithm, which applies genetic operators to search in the space of all possible process models, can successfully deal with the aforementioned challenges. In this paper, we reduce the computation time by using a distributed setting. The population is distributed between the islands of a computer network (e.g. a grid). To further accelerate the method we use sample-based fitness evaluations, i.e. we evaluate the individuals on a sample of the event log instead of the entire event log, gradually increasing the sample size if necessary. Our experiments show that both sampling and distributing the event log significantly improve the performance. The actual speed-up is highly dependent of the combination of the population size and sample size.

**Keywords:** Genetic algorithms, business intelligence, process mining, sampling.

## 1 Introduction

Process mining has emerged as a discipline focused on the discovery of process models from event logs. Event logs can be seen as a collection of process executions records, i.e. *traces*, each related to a *process instance*. The process models usually graphically depict the flow of work using languages as Petri Nets, BPMN, EPCs or state machines. The discovered process model should be able to reproduce most of the traces from the event log and "not too many" traces which are not present in the log. Real life case studies performed for, e.g., Phillips Medical Systems [9] and ASML [13] have shown that process mining algorithms can offer insight into processes, discover bottlenecks or errors and assist in improving processes.

Most of the Process Mining Algorithms (PMAs) [2,16] use heuristic approaches to retrieve the dependencies between activities based on patterns in the logs. However, these heuristic algorithms fail to capture complex process structures and they are not robust to noise or infrequent behavior. In [3] Alves de Medeiros proposed a *Genetic Mining Algorithm* (GMA) that uses genetic operators to overcome these shortcomings. The GMA evolves populations of process models towards a process model that fulfills the predefined fitness criteria. Fitness criteria quantifies the ability of the process model to replay all the behaviors observed in the log without allowing additional ones.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 224–237, 2011.

An empirical evaluation [4] shows that the GMA indeed achieves its goal and discovers better models than other PMAs.

Although GMA prevails against other algorithms in terms of model quality, heuristic-based algorithms proved to be significantly more time efficient [4]. In [5], we proposed a coarse grained approach [15] called the *Distributed Genetic Miner Algorithm* (DGMA) that improves the GMA time consumption by dividing the population into subpopulations distributed over grid nodes (called *islands*) and exchanging genetic material. The experimental results show that the DGMA outperforms always the GMA.

In this paper, we enhance our method by using *sample-based fitness evaluation*. The idea is that each subpopulation evolves guided by a *random sample* of traces from the event log. We create the initial subpopulation using a smart and fast heuristics based on the whole log and then we use a random sample of the log for fitness computations until the desired fitness is achieved on this sample. The use of a larger sample increases the chance that this sample is representative for the whole log, i.e. we achieve the desired fitness on the whole log as well. However, a larger sample size increases the fitness computations time and thus reduces the time advantage of sampling. To balance between the two objectives (mining quality and time efficiency) each island uses an *iterative sample-based algorithm* (denoted *SGMA*) that adds a new part to the sample until the discovered process model has the required quality for the entire log. After increasing the sample, we use the already mined population of process models as initialization. In order to increase the chance that the initial sample is representative enough for the whole log, we use different initial samples for each of the subpopulations.

We empirically assess the convergence of several event logs with different characteristics. We analyze the combined effect of the sample size and subpopulation size. Our results demonstrate that a proper balance between the subpopulation size and the sample size needs to be found in order to obtain the best speed-ups.

**Related Work.** Sampling is a common practice in business intelligence domains such as data mining and knowledge discovery. Kivinen and Manilla [12] use small amount of data to discover rules efficiently and with reasonable accuracy. Tan [14] discusses the challenges in using sampling such as choosing the sample size and also proposes to increase the sample size progressively.

Reducing the fitness computation time is one of the main topics in *genetic algorithms* literature such as: approximating the fitness function by a meta-model or a surrogate [10,11], replacing the problem with a similar one that is easier to solve [11], or, inheriting the fitness values [7]. The techniques used in [8] are close to the one we use, although their motivation and goals are different: they analyze the effect of sampling when the fitness function is noisy. Fitzpatrick and Grefenstette [8] show that genetic algorithms create "more efficient search results from less accurate evaluations".

The paper is organized as follows: Section 2 introduces the process mining as a techniques to extract process models form event logs; Section 3 presents the distributed solution and its implementation. We analyze the performance of our solution for different parameters combinations in Section 4. We give conclusions and describe future work in Section 5.

**Table 1.** A simplified event log

| Process instance id | Trace |
|---|---|
| 1 | A B C |
| 2 | A D B G C |
| 3 | A B E F C |
| 4 | A D B G E F C |
| 5 | A B E F E F B E F C |
| 6 | A D B G E F B E F E F C |

**Table 2.** Causal Matrix

| Activity | Input set | Output set |
|---|---|---|
| A | - | B ∨ D |
| B | A ∨ D ∨ F | C ∨ E ∨ G |
| C | B ∨ F ∨ G | - |
| D | A | B ∧ G |
| E | B ∨ G ∨ F | F |
| F | E | B ∨ C ∨ E |
| G | B ∧ D | C ∨ E |

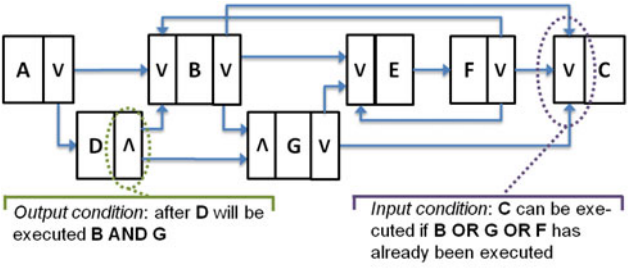



**Fig. 1.** Process model representing the behavior in the event log from Table 1

## 2 Process Mining and Event Logs Characteristics

Process mining aims to discover process models from event logs, thus recording (parts of) the actual behavior. An *event log* describes previous executions of a process as sequences of *events* where each event refers to some activity. Table 1 presents a simplified event log inspired by a real-world process: a document issue process for a Dutch governmental organization. This log contains information about six process instances (individual runs of a process). One can notice that each process instance is uniquely identified and has an associated *trace*, i.e., the executed sequence of activities. The log shows that seven different activities appear in the log: *A, B, C, D, E, F,* and *G*. Each process instance starts with the execution of *A*, ends with the execution of *C* and contains *B*. Process instances with id 2, 4 and 6 also contain *D* and *G* suggesting that there is a relation between their occurrences. *E, F* and *G* appear always after *B* and *D* occurs prior to *B*. Moreover, the occurrence of *F* is always preceded by the occurrence of *E*. Instances 5 and 6 show that loops are possible in the process.

Different traces of an event log might contain information about certain dependencies between activities which can be deducted from other traces. In Table 1, trace 6 does not add any information about the process structure to traces 1-5. For example, both loops *BEF* and *EF* in trace 6 can be identified from trace 5. Note that there are multiple models that can reproduce the same event log. The quality of a model is given by its ability to balance between underfitting and overfitting. An underfitting model allows for too much behavior while an overfitting model does not generalize enough.

Figure 1 shows a process model with fitness 0.98 for the example event log from Table 1. The process model is a graph model that expresses the dependencies between activities. Each node in the graph represents an activity. Each activity has an *input* and *output set*. *Causal matrices* [3] are used to represent the dependencies between activities in a compact form. The causal matrix for the process model in Figure 1 is shown in Table 2. Since A is the start activity, its input condition is empty and A is enabled. After the execution of A the output condition $\{B \vee D\}$ is activated. Further on, B is enabled because the input condition of B requires that at least one of the activities A, D or F is activated before B. The input condition of D requires only A to be activated before D. If we assume that D is executed, we observe that B is automatically disabled because the output condition of A is not longer active. After D is executed, B is enabled because the output condition of D, {B}, is activated. For more details on the semantics of causal matrices we refer to [3,4].

The computational complexity of a particular PMA depends on the log characteristics. The following parameters give the basic characteristics of a log: the *size* of a log (the sum of the lengths of all traces); the *number of traces* (influencing the confidence in the obtained process models); and the *number of different activities* (defining the search space for the model). In our example, the event log size is 42, the number of traces is 6 and the number of different activities is 7.

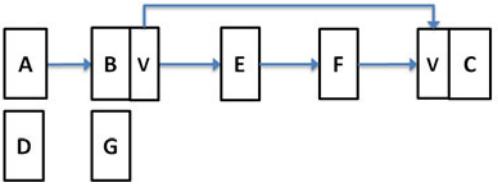
Many PMAs, $\alpha$-algorithm [2] are linear in the size of the log. However, such algorithms do not perform well on real-life data [9,13] and therefore more advanced PMAs are needed. For example, the $\alpha$-algorithm does not capture the dependency between the activities $D$ and $G$ from the example event log which results in underfitting the event log. *GMA* [3,4] applies genetic operators on a population of process models in order to converge to models that represent the behavior in the event log more precisely. The main advantages of GMA are the ability to discover non-trivial process structures and its robustness to noise as demonstrated by [4]. Its main drawback is the time consumption, due to the two factors: 1) the time required to compute the fitness for an individual, and 2) the large number of fitness evaluations needed.

## 3 Distributed Sample-Based Genetic Mining Algorithm (DSGMA)

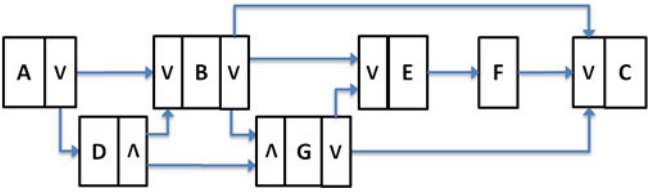
In this section, we present our distributed sample based approach for GMA. The distribution uses a coarse-grained approach. In the *coarse-grained* approach the population is split into subpopulations. Each subpopulation resides on an *island* and the *coordinator* orchestrates the islands.. We first present the island algorithm and then the distribution architecture. Each subpopulation runs independently and individuals are exchanged among subpopulations via *migration*.

### 3.1 Island Sample Based Genetic Process Mining Algorithm (SGMA)

In this subsection we show how sampling can accelerate GMA on one island. The island *Sample-based Genetic Mining Algorithm* (SGMA) is based on [3,4]. The algorithm improves the overall execution time of the existing GMA by incrementally learning from samples of traces from the event log. Algorithm 1 presents the main SGMA steps. SGMA uses the same genetic operators and fitness computation as GMA [3,4].




(a) Results after the first iteration. Fitness on the sample is 0.98; fitness on the log is 0.77




(b) Results after the second iteration. Fitness on the sample is 0.98; fitness on the log is 0.87

**Fig. 2.** Process models for the example event log (Table 1)

The main difference is in handling input data. The algorithm starts with *PartitionTheLog*(*Log*, $Log_1$, ... , $Log_n$) that divides the *Log* into *n* random samples of "equal" size. At each *iteration m*, genetic operators are applied using a sample from the event log corresponding to the union of the first *m* samples. The iteration stops when *StopCondition*, i.e. reaching a given quality of the population, is fulfilled. Note that the *StopCondition* of the internal *while* loop is evaluated on the *SampleLog* and not on the entire *Log*. The algorithm *stops* when the same *StopCondition* is satisfied for the *entire event log* or is activated by the coordinator.

The SGMA individuals are graph models, such as the one presented in Figure 1. The individuals are encoded as a set of activities and their corresponding input and output sets, i.e., a *causal matrix*. Note that it is trivial to construct the graph representation from Figure 1 based on the compact representation from Table 2. Each process model contains all the activities in the log; hence, the search space directly depends on the number of activities in the log.

*BuildInitialPopulation(*Log*)* generates individuals from the search space using an heuristic approach. This heuristic uses the information in the log to determine the probability that two activities have a dependency relation between them: the more often an activity *A* is directly followed by an activity *B*, the higher the probability is that the individuals are built with a dependency between *A* and *B*.

*ComputeFitness*(*P*, *(Sample)Log*) assesses each individual against the (*Sample*)*Log*. *Fitness* values reflect how well each individual represents the behavior in the log with respect to *completeness*, measuring the ability of the individual to replay the traces from the log, and *preciseness*, quantifiying the degree of underfitting the log. The *completeness* of an individual is computed by parsing the log traces. When an activity of a trace cannot be replayed, i.e. the input condition is not satisfied, a penalty is given. In order to continue, the algorithm assumes that the activity was executed and it tries to parse

**Input**: *Log*, *StopCondition*, *n* `// number of samples`
**Output**: *ProcessModel*
*PartitionTheLog*(*Log*, $Log_1$, ... , $Log_n$);
*P* = *BuildInitialPopulation(*Log*)*;
*SampleLog* = {}; *m* = 1;
**repeat**
  *SampleLog* = *SampleLog* ∪ $Log_m$;
  *Fitness* = *ComputeFitness*(*P*, *SampleLog*);
  **while** StopCondition*(*Fitness*)*==*false* **do**
    *P* = *ComputeNextPopulation(P,Fitness)*;
    *Fitness* = *ComputeFitness*(*P*, *SampleLog*);
  **end**
  *m* = *m* + 1;
  *Fitness* = *ComputeFitness*(*P*, *Log*);
**until** StopCondition(Fitness)==*true* ;
*ProcessModel* = *bestIndividual*(*P*);

**Algorithm 1.** Island algorithm - SGMA

the next activity from the trace. Additional penalties are given when the input/output conditions enable an activity incorrectly, e.g. if in the input condition, activities A and B are in an AND relation and in the trace only B is executed. The fitness quantifies all the penalties and compares them with the number of correctly parsed activities. The fitness *preciseness* penalizes the individuals that allow more behavior than their siblings. The exact formula is out of the scope of this paper but can be found in [3,4].

*ComputeNextPopulation(P, Fitness)* first checks if the island received new individuals. New individuals are integrated into the population according with the migration policy. We present the possible migration policies in Subsection 3.2. Then, the genetic operators (*selection*, *mutation* and *crossover*) are applied to generate a new population. The selection operator ensures that the best individuals are carried forward from the current population to the next one. The mutation modifies the input/output conditions of a randomly selected activity by insertion, removal or exchanging the AND/OR relations of the activities. The crossover exchanges the input/output conditions of a selected activity between two individuals.

The convergence of the SGMA algorithm is ensured by the fact that at each iteration a larger sample is taken together with the convergence of the traditional GMA. If new activities or dependencies appear in the newly added sample, they are discovered by exploring the space using the genetic operators. Basically, the algorithm is learning at each iteration by increasing its knowledge. Figure 2 shows one variant of intermediary process models when SGMA is applied to the example event log (Table 1). We consider the log divided in three subsets: {1, 3}, {2, 4}, and {5, 6}. Figure 2a is based on {1, 3} and does not contain D and G connected, which does not harm the fitness value since the activities do not appear in the first sample. At the second iteration, when using the first and the second samples, D and G are integrated into the process model but the loops are not present. After the last iteration we obtain the process model presented in Figure 1. We observe that the algorithm converges by adding new dependencies to the previously obtained model.

The SGMA exploits the *redundancy* present in most of the event logs. By redundancy, we understand that many traces contain information which may be derived from other traces. For the event log in Table 1, the same process model is obtained if the log would only contain traces 1-5. Sources of redundancy are independent choices, loops, various interleavings, etc. For example, there is no need for having all the possible interleavings in order to discover the concurrency. The redundancy of an event log becomes visible when comparing the sample size that ensures SGMA convergence after one iteration to the log number of traces. Obviously, *the more redundant the log is, the more efficient SGMA is in comparison with GMA*.

The efficiency of SGMA is also influenced by other factors, such as the population size per island and the complexity of the log. A known result in genetic algorithms (GA) [15] is that a GA is more efficient if the population contains more diversity. In the case of SGMA, a problem can be the specialization of the overall population towards the sample when the sample contains mostly atypical traces for the event log, or is too biased towards some subset of traces. In this case, the SGMA will spend a significant amount of time to discover features of the traces added in the following iterations. *Lower population sizes have more chances to specialize towards the sample and thus making SGMA less efficient.*

Experiments performed in [5] showed that for every log there exists a threshold for the mean number of fitness computations (MTFC) that have to be performed such that (D)GMA converges. MTFC depends on the complexity of the event log from a mining point of view, i.e. how many complex patterns [1], e.g., loops and parallel branches, the process model generating the log contains. The more difficult a log is, the higher MTFC is. *Since SGMA reduces the fitness computation time by using a sample of traces from the event log, the more difficult a log is, the more efficient SGMA is.*

### 3.2 Distribution Architecture

Our DSGMA distributes the work using the coarse-grained approach [6,15]. The architecture is composed of one *coordinator* and a number of *islands*. Figure 3 presents the overall architecture.

The coordinator initializes the islands by setting the initial parameters and to orchestrate the migration of individuals between the islands. In the initialization phase the coordinator sends the log and the parameter values to each island. The islands run SGMAs using the *same* parameters (i.e. crossover rate, mutation rate and elitism rate). In order to increase the chance that the initial sample is representative enough for the entire log, we *initialize each island with a different initial sample*. The islands send information regarding their current state, such as best and average fitness, to the coordinator that uses this information to coordinate the migration or to stop the algorithm.

Note that the islands do not synchronize on the receiving/sending of the individuals. Islands may receive individuals at any moment of time and they integrate them in the current population according to the migration policy. The migration policy is defined by the following parameters: *Integration Policy* (IP) (how subpopulations integrate the received individuals), *Selection Policy* (SP) (type of individuals sent when the migration takes place), *Migration Interval* (MI) (number of generations between consecutive migrations), and *Migration Size* (MS) (percentage of the population sent in

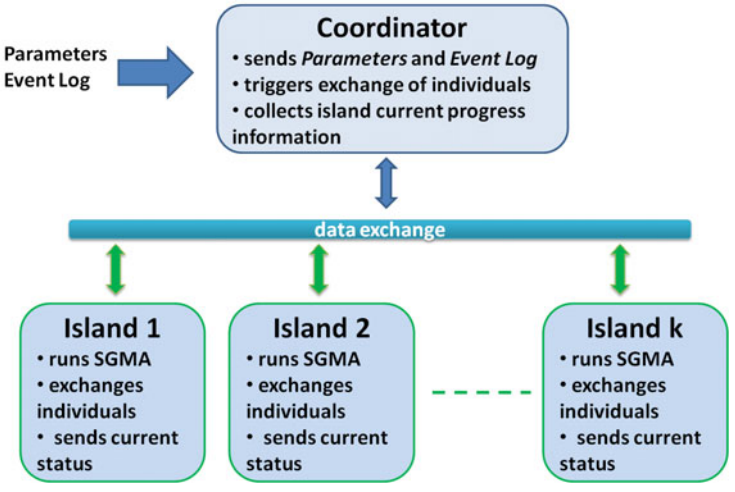



**Fig. 3.** DSGMA Architecture

every migration step). Based on our previous results [5] obtained for the distributed genetic miner and other distributed genetic algorithm results [6,15] we selected the following parameter settings for the migration policy: IP is "replace worst individuals with the migrants"; SP is to send the best individuals; MI equals 10 generations, and MS equals 10% of the subpopulation size.

The distribution parameters are the *Number of Islands* (NI) (subpopulation) and the *Population Size* (PS) per island. In [5], we show that the best speed-up of DGMA is obtained for PS equal to 10. However, by using only a sample to guide the genetic operators we obtain better speed-ups for higher PSs. The reason is that if the initial sample is not representative, the following iteration needs diversity in the population in order to converge. If this is not the case the diversity needs to be created by mutation, and hence the DSGMA will result in very long execution times. In the next section, we analyze the combined effect of PS and the Initial Sample Size (ISS).

The SGMA and the DSGMA are implemented as part of the ProM framework. The ProM framework (www.processmining.org) is an open source Java-based plugable framework that has shown to be highly useful on many real-life case studies. The next sub-section details this implementation.

### 3.3 Implementation

The GMA and the DGM are implemented as part of the ProM framework. The ProM framework[1] is an open source Java-based plugable framework that has shown to be highly useful on many real-life case studies. We implemented the DGM as a collection of plug-ins:

1. The *IterativeGeneticMiner* plugin that implements the iGMA.

[1] See www.processmining.org for details and for downloading the software.

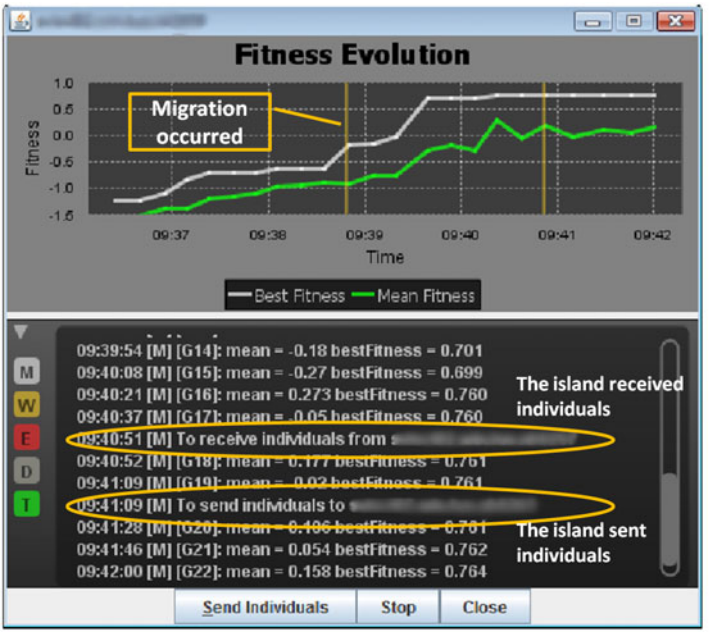



**Fig. 4.** Islands progress visualization in ProM

2. The *DistributedIterativeGeneticMinerIsland* plugin implements the island algorithm. The plugin communicates with the master and other islands via a TCP/IP communication. The *GeneticMiner* plugin is called for the evolution steps of the subpopulation.
3. The *DistributedIterativeGeneticMinerMaster* plugin takes as input an log, a list of IP addresses and the values of the parameters. The master triggers via a TCP/IP communication the start of island plugins on remote ProM frameworks hosted at the given IP addresses. Each island is started with the same parameter settings. The parameters are split into two sets: iGMA parameters and migration parameters. The plugin supports the definition of different migration policies and communication strategies.

We implement the communication between different ProM frameworks using sockets[2] technology. Each object (e.g. logs, individuals) is first translated to an XML based language and then sent using TCP/IP. The migration of individuals is made peer to peer between islands, the master just triggering the exchange. The sending/receiving of individuals is done asynchronously in different threads, i.e. in parallel with the main GMA. The receiving of individuals is acknowledged via shared objects between the threads.

The *DistributedGeneticMinerMaster* plugin enables the user to monitor the progress of each island and to interact by triggering the migration or stop the evolution. Figure 4 shows a snapshot of an island progress window. The vertical lines in the graph mark the migration moments.

[2] http://java.sun.com/docs/books/tutorial/networking/sockets/

**Table 3.** Logs settings

| Name | Number of activities | Number of traces | Size |
|---|---|---|---|
| Log A | 24 | 229 | 3994 |
| Log B | 25 | 300 | 9054 |
| Log Municipality | 18 | 1394 | 11467 |

## 4 Experimental Results

In this section we evaluate the effect of the population size and sample size on the execution time of the DSGMA for three different logs: *A*, *B* and *Heusden*. The first two logs are generated by students as part of their assignment for the process mining course. The third log is a real life log for the process of handling objections against the real-estate property valuation at the Municipality of Heusden.

Table 3 presents the *number of activities*, *number of traces* and *size* of the logs. Logs *A* and *Heusden* have rather a simple structure, which makes them "easy" to mine. Log *B* has a more complex structure, which makes it a difficult mine. The challenge of log *Heusden* is in the large number of traces. In the fitness computation, each individual is assessed against all the traces in the log, which implies longer fitness computation time per individual for *Heusden*.

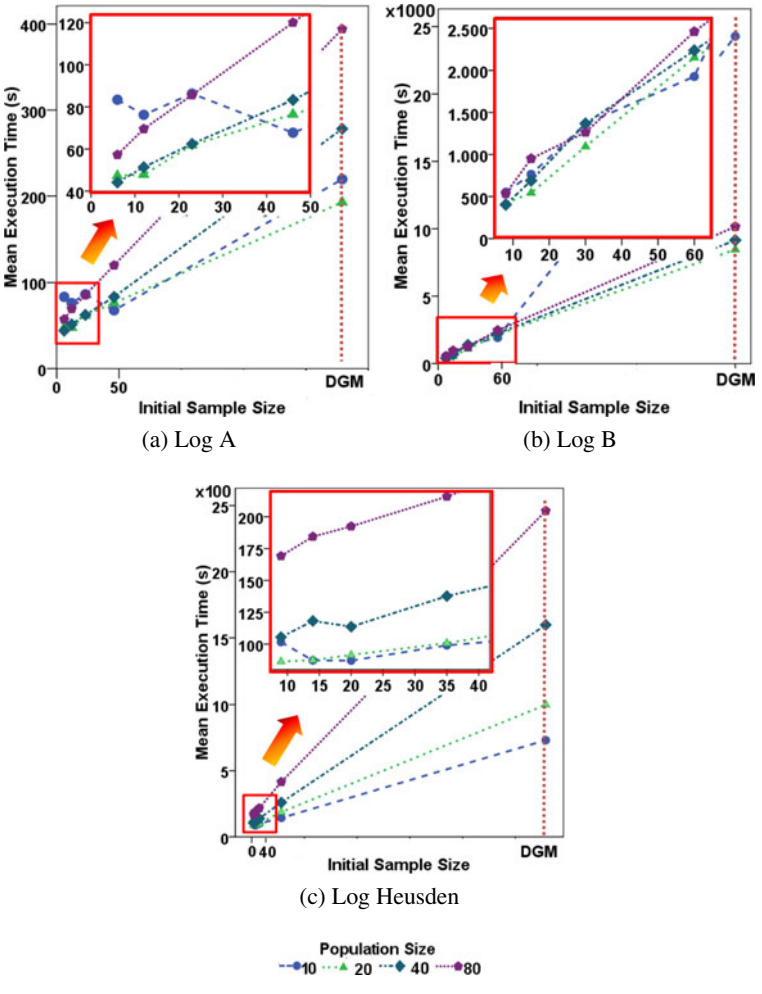
We use a testbed with the following configurations: eight Intel(R) Xeon processors running at 2.66 GHz and using 16 Gb RAM. We use eight islands each running on of one of the eight processors. Since islands are identical from the performance point of view, we assess the results in terms of the *Mean Execution Time* (MET) needed to find an individual with a fitness higher than 0.9. Note that the highest possible fitness value is 1. The execution time is averaged over 20 independent mining executions. For the visualization and analysis of results we use SPSS (http://www.spss.com).

In [5], we showed that by using DGMA, a coarse grained distribution of the GMA, the execution time is reduced significantly. For eight islands, DGMA mines logs A and Heusden four times faster than GMA and log B is mined almost eight time faster than GMA. In this work, we compare DSGMA results with DGMA, so we investigate the added value of sampling.

Our experiments focus on the influence of the the *Population Size per island* (PS) and the *Initial Sample Size* (ISS). Figure 5 shows the results obtained for the three logs. Note that we increase the ISS until the algorithm terminates after one iteration for all of the 20 repetitions. It is easy to observe that DSGMA is more efficient than DGMA.

In Subsection 3.1 we identified three factors that influences the efficiency of (D)-SGMA: 1) the diversity in the population; 2) the degree of redundancy in the event log, and 3) the difficulty of the event log.

In Figure 5, we observe that the algorithm converges, even with very small ISS (less than 10 traces in the initial sample) due to its iterative nature. For $PS = 10$ the MET is higher, especially for small ISS, than for $PS > 20$ because the population looses its diversity and specializes towards the initial sample. When the ISS is large, the required number of iterations decreases, so does the risk of a biased specialization towards the sample, and small PS (e.g. PS = 10 for log Heusden) gives good results. Note that the combination of low ISS (i.e., less than 10 traces) and PS = 40 for logs A and B and




(a) Log A (b) Log B

(c) Log Heusden

**Fig. 5.** Execution Time for different Population Sizes and different Initial Sample Sizes

**Table 4.** Comparison of DGMA and DSGMA using the three logs. The last column shows the speed-up of DSGMA compared to DGMA.

| Name | Type | PS | ISS | MUNT | MFC | MET | Speed-up |
|---|---|---|---|---|---|---|---|
| A | DGMA | 20 | 229 | 229 | 1002 | 194 | 4.4 |
| | DSGMA | 40 | 6 | 12.6 | 2184 | 44 | |
| B | DGMA | 20 | 300 | 300 | 14372 | 8467 | 20.9 |
| | DSGMA | 40 | 8 | 18 | 19732 | 405 | |
| Municipality | DGMA | 10 | 1394 | 1394 | 808 | 731 | 8.5 |
| | DSGMA | 20 | 9 | 36.9 | 3096 | 86 | |

PS = 20 for log Heusden results in the best MET. We can conclude that DSGMA produces better speed-ups for small ISS and "medium" PS.

The last two factors are further emphasized in Table 4 that presents the results in terms of MET, *Mean Used Number of Traces* (MUNT) and *Mean Number of Fitness Computations* (MFC) for the configurations creating the best METs for DGMA and DSGMA. We observe that the difficulty of an event log has more influence than the log redundancy: log B has the highest speed-up due to the higher number of MFC. When the logs complexities are comparable, such as for logs A and Heusden, the degree of redundancy influences the speed-up difference: DSGMA for log Heusden requires only 2.8% of the traces in order to converge and thus has a higher speed-up than DSGMA for log A that requires 5.5% of the log traces.

Note that MNT for log A is 10% of the log size and the speed-up is 4, for log B MNT is 20% and speed-up is 20, and for log Heusden MNT is 5% and the speed-up is 9. The reason that log B has a higher speed-up than the other two despite of a higher MNT is that DSGMA for log B converges in more than 400 generations and DSGMAs for logs A and Heusden converge in less than 40 generations. More generations are required for DSGMA to converge then more fitness fitness computations are performed and thus, higher speed-ups are achieved.

The speed-up, computed as the ratio between the minimum MET for DGM and the minimum MET for DSGMA is: 4 for log A, 20 for log B and 9 for log Heusden. The low value for log A derives from its fast convergence and the log redundancy, i.e. less than 10% of the traces are necessary for the initial sample. Even if log Heusden converges almost as fast as log A, we obtain a better speed-up because less than 5% of the log is needed to ensure the convergence. In the case of log B, MET of DSGMA is *20 times faster* than DGM due to its slow convergence (in more than 400 generations) despite the higher necessary number of traces needed for the convergence, i.e. more than 20% from the log size.

## 5 Conclusions and Future Work

Genetic process mining algorithms succeed to find better models at the price of longer computation times. In this paper, we proposed a a sample-based distributed algorithm for GMA that significantly improves its execution time. Moreover, we conducted

empirical evaluations of the effect of the population size and initial sample size on the performance of the algorithm.

The experimental results showed that DSGMA significantly speeds up the computation. The algorithm performs better for difficult process mining tasks such as log B. Our experiments demonstrate that best speed-ups are obtained when the initial sample is rather small (less than 10 traces) because the algorithm learns an initial approximation of the model fast. The tuning of this initial model is afterwards done by learning from the additional traces. The only concern is whether the population available at the start of a new iteration contains enough diversity to learn fast enough from the traces added later. A "medium" PS (20 for log Heusden and 40 for logs A and B) reveals to be optimum based on the experiments performed.

Since representativity of the samples highly influences the speed-up, we are currently investigating "smart" sampling strategies that will ensure a given degree of representativity for the initial sample.

## References

1. van der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14(1), 5–51 (2003)
2. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
3. Alves de Medeiros, A.K.: Genetic Process Mining. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands (2006)
4. Alves de Medeiros, A.K., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic process mining: An experimental evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
5. Bratosin, C.C., Sidorova, N., van der Aalst, W.M.P.: Distributed genetic miner. In: Proceedings of the 2010 IEEE World Congress on Computational Intelligence (IEEE CEC 2010), Barcelona, Spain, July 18-23, pp. 1951–1958. IEEE, Los Alamitos (2010)
6. Cantú-Paz, E.: A survey of parallel genetic algorithms. Calculateurs Paralleles, Reseaux et Systems Repartis 10(2), 141–171 (1998)
7. Chen, J.-H., Goldberg, D.E., Ho, S.-Y., Sastry, K.: Fitness inheritance in multi-objective optimization. In: GECCO, pp. 319–326 (2002)
8. Fitzpatrick, J.M., Grefenstette, J.J.: Genetic algorithms in noisy environments. Machine Learning 3, 101–120 (1988)
9. Günther, C.W., Rozinat, A., van der Aalst, W., van Uden, K.: Monitoring deployed application usage with process mining. Technical report, BPM Center Report BPM-08- 11, BPMcenter.org (2008)
10. Jin, Y.: A comprehensive survey of fitness approximation in evolutionary computation. Soft Computing 9(1), 3–12 (2005)
11. Jin, Y., Branke, J.: Evolutionary optimization in uncertain environments-a survey. IEEE Trans. Evolutionary Computation 9(3), 303–317 (2005)
12. Kivinen, J., Mannila, H.: The power of sampling in knowledge discovery. In: PODS 1994: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 77–85. ACM, New York (1994)

13. Rozinat, A., de Jong, I.S.M., Günther, C.W., van der Aalst, W.M.P.: Process mining applied to the test process of wafer scanners in asml. IEEE Tran. on Syst., Man, and Cybernetics 39(4), 474–479 (2009)
14. Tan, P.-N., Steinbach, M., Kumar, V.: Introduction to Data Mining. Addison-Wesley Longman Publishing Co., Inc., Boston (2005)
15. Tomassini, M.: Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time. Springer-Verlag New York, Inc., Secaucus (2005)
16. Weijters, A.J.M.M., van der Aalst, W.: Rediscovering workflow models from event-based data using little thumb. Integr. Comput.-Aided Eng. 10(2), 151–162 (2003)

# FaDe: RESTful Service for Failure Detection in SOA Environment*

Jerzy Brzeziński, Dariusz Dwornikowski, and Jacek Kobusiński

Institute of Computing Science,
Poznań University of Technology, Poland
{jbrzezinski,ddwornikowski,jkobusinski}@cs.put.poznan.pl

**Abstract.** FaDe service is a novel proposal of failure detection service based on REST paradigm for SOA environment. It is fully distributed service. Internal communication between FaDe nodes can be realized using two communication protocols (gossip, Kademlia). It makes the nodes cooperation efficient and provide good level of scalability. The failure monitoring is based on accrual failure detector which provides flexibility in the context of different client expectations considering the speed and accuracy of detections.

**Keywords:** failure detection, distributed systems, service-oriented architecture.

## 1 Introduction

The use of distributed systems as a computing platform constitutes a very promising research and business area due to their availability, economic aspects and scalability. The intense development of Grids, P2P networks, cluster and high-speed network topologies gave the possibility to allocate an enormous amount of resources to distributed applications at a reasonably low cost. Their level of parallelism can improve the performance of existing applications and raise the processing power of distributed systems to a new higher level. Unfortunately, those systems are failure prone and the probability that a failure occurs during computations is higher than in traditional systems.

Therefore, to overcome the problem one should construct a fault tolerant mechanism to detect unavoidable failures and makes their effects transparent to a user. Failure detection mechanism is one of the key component needed to provide fault tolerance. Chandra and Toueg presented a concept of failure detectors [1], as an abstract mechanism that supports asynchronous system model. A failure detector that makes no mistakes and eventually suspects all failed processes is called *perfect failure detector*. However, this requirements are very difficult to be fulfilled in practice due to asynchronous nature of real systems. So, one can

* The research presented in this paper was partially supported by the European Union in the scope of the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 238–243, 2011.

weaken safety (completeness) or liveness (accuracy) property to construct *unreliable failure detector*, which can be used to detect failures in an asynchronous system [10].

Service-oriented architecture software is an increasingly popular model of software [3] that provides separation of some independent components (services), which can cooperate with each other through well defined interfaces. Scalable Web service FADE, which allows a very flexible failure detection of individual nodes, services running on them, or the unavailability of resources provided by these services is presented below. The proposed solution takes into account various aspects, both those concerning the environment in which failure detection will be performed, as well as those relating to client expectations relative to this type of service.

The remaining part of the paper is structured in the following way. In Section 2 general concept of FADE service is presented. Section 3 describes the architecture of the FADE service node. In Section 4 main aspects of failure detection module are discussed. Brief description of communication mechanism used by the service is presented in Section 5. Related work is described in Section 6. Finally, Section 7 brings concluding remarks and summarizes the paper.

## 2 FADE General Concept

An independent service that provides functionality involving failures detection in a distributed environment can be used by different clients without having to re-implement it in every single individual application. Such an approach would also allows more efficient use of available resources like network bandwidth or power computing.

The independent service must assume the existence of clients with varying preferences for speed and accuracy of failure detection. This means that a simple binary response (crashed/correct) will not always be adequate to meet their demands. In this context, a concept in which the mechanism for monitoring services will be based on incremental failure detector proposed in [2] was adopted. It gives the possibility of transferring responsibility for the interpretation of the result to the client, which, depending on its individual preferences to decide by itself whether the answer returned by the service means a failure or not.

Another important assumption is the one concerning the ease of setup and adjustment the service to the environment in which it operate. WAN environment characterized by high dynamics and variability will have an impact on the service efficiency. Therefore, the service itself should be constructed in such a manner that its configuration and tuning to the existing conditions will be relatively easy and automatic.

In contrast to the services running on the local network, which by definition are limited to a certain area, distributed services dedicated to wide area network must take into account the possibility of their expansion over a wide geographical area. In this context, scalability is a key element that must be considered.

As stated in the introduction, a distributed environment is prone to various types of failures. If a failure detection service operating in such an environment is

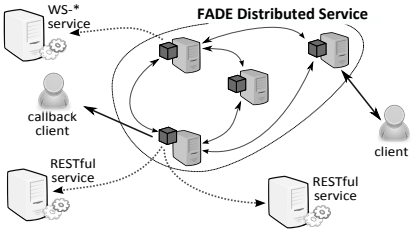



**Fig. 1.** FADE connection topology

to be element that provide reliability for the application , it should not introduce additional risks associated with its susceptibility to its own failure. Mechanisms that can reduce this risk are redundancy of key components and their decentralization. Responsibility dispersal of monitoring the individual components will minimize the effect of so-called "single point of failure".

Figure 1 presents a FADE service connection topology. In this case, the FADE service consists of four nodes. They are connected and communicate with each other in order to exchange information about connection topology and monitored service states. The FADE nodes monitor three services, both WS-* compliant and RESTful [4]. It is transparent to the client, which node monitors particular service. Moreover one can distinguish different type of clients standard and callback which use FADE service in different manner. In general, clients can inquire FADE service about the status of the monitored services. Depending on the type of request, appropriate action is performed.

## 3 FADE Node Architecture

The modular architecture of FADE node allows easy modification of every single component, or even the substitution to a new one that provides compatible interface.

The *main* component task is the integration and interpretation of data supplied by other components. One of the main action performed by this component is interpretation of monitoring results and client request. It is also responsible for access to resources. The *monitoring* component is directly responsible for monitoring the services. It works by exchanging messages with the these services in a manner specified in the configuration with the specified frequency. Based on the results of this message exchange the state of the monitored services can by determined. *Internal communication* component is the module that is responsible for communication with other FADE nodes that form a distributed failure detector. Its purpose is to exchange information that refer to both the state of monitored services and internal configuration of FADE service. *Client interaction* component is the one that provides the external interfaces compliant to WS-* standards and REST paradigm. Despite the fact that the FADE service is dedicated to monitoring RESTful Web services, it also allows to monitor

services built in accordance with the WS-* standards that use SOAP protocol. Moreover, it provides SOAP compliant interface. Finally, the *configuration* component manages the service setup process. It implements mechanism that can accept various source of configuration data: command line, file, remote call, and setup the service node accordingly.

## 4 Failure Monitoring

A widely used method to perform failure detection is a heartbeat technique. This method is based on periodical probing other nodes in order to determine their status or signaling own status to them. This popular technique is based on the assumption about the maximum delivery time and uses timeouts to decide which node should be suspected as the crashed one. Depending on the way the exchange of messages is initialized one can distinguish two types of heartbeating: the monitored object continuously advertises its state or the failure detector asks object about its state. The latter approach allows a better control of the monitoring and also is more appropriate in the case of a failure detection service, which should be an active part in the monitoring process.

*Failure Detector Mechanism.* A typical binary model of failure detector response has serious limitation when considering failure detection as a generic service. Those restrictions come from the fact that there are different expectations of detection time and accuracy. Some user applications require aggressive detection even at accuracy cost, while others are interested in a more conservative detection with low level of false suspicions instead of fast but inaccurate decisions. By using binary response generated by a failure detection service, it is generally impossible to fulfill these requirements. It is because the timeout threshold that marks the line between the correct and crashed process should be different. This problem of contradictory requirements can be solved by using accrual failure detector [2]. It assumes delegation of the decision about object crash to the end user application. The FaDe provides only non-negative real value representing current suspicion level that should be further interpreted by the application. Since the detection quality undertaken using this mechanism is dependent on the characteristics of the network environment in which the monitoring process is being implemented, FaDe service also offers a wide range of optional parameters, which allow to adjust this process.

*Interactions with Clients and Services.* FaDe service offers three different methods to obtain information about the state of monitored services: *standard*, *callback* and *ad-hoc* query. Service client chooses one of them according to its own expectations and preferences. These methods complement each other, creating a complete and consistent interface. Standard query is a basic method for obtaining information from the FaDe service. It assumes that a client orders some service monitoring. Since then, the service is continuously monitored by FaDe and at any time, any client may request information about the state of this service by sending standard query to any FaDe node. Continuous polling FaDe

service about the status of monitored service is not always the best solution. Often, clients require only information about state changes. Such a scenario can be realized through a callback queries. Client registers callback request and depicts event it is interested in. If such an event takes place, the client is informed by the FaDe service. In some situations, the client is only interested in one-time information on a service state as soon as possible. For this purpose, a mechanism of ad-hoc queries has been made available. These queries are handled by FaDe in a different manner than the other two described earlier. Verification of the service availability is carried out in a simplified manner, consisting in sending a single message monitoring.

## 5 Internal Communication

It is sufficient to run only one FaDe node to make the service fully available for clients. However, the full potential of the service is revealed only in the case of multiple cooperating nodes. This means that there must be efficient and reliable mechanism for exchanging information between nodes. It should take into account the characteristics of the runtime environment: local and wide area networks. Thus, the service provides two alternative communication modules. The first one uses a probabilistic approach and is based on the idea of epidemic protocols [5]. The second one is an adaptation of the Kademlia protocol [8] used in P2P networks. FaDe service administrator can choose one of them depending on the characteristics of the network, its own preferences and available resources. Both modules use randomized communication pattern which increase fault tolerance to broken links and network failures. The choice of the internal communication protocol is transparent to the service client.

## 6 Related Work

However FaDe is the first proposal of RESTful failure detection service, there are frameworks for failure detection in SOA environment that uses SOAP protocol. FT-SOAP [7] and FAWS [6] are the solutions that consist of a group of replica services that are managed by external components. FTWeb [11] has components that are responsible for calling concurrently all the replicas of the service and analyzing the responses processed before returning them to the client. Thema [9] is an example of framework that can tolerate Byzantine faults. Finally, Lightweight Fault Tolerance Framework for Web Services [12] achieve fault tolerance through consensus-based replication algorithm.

## 7 Conclusion

In conclusion, presented FaDe service allows the detection of failures in a distributed service-oriented environment. Thanks to compliance with the REST architectural style, the service is lightweight and use available network resources

sparingly. Due to resource-oriented approach, its interface is simple, clear and allows easy integration with other services. The ability to accept SOAP requests as well as the ability to monitor not only RESTful services makes the FaDe service a very versatile proposal. By using mechanism based on the accrual failure detector concept service provides flexibility in the context of client expectations related to the accuracy and time of failure detection.

## References


1. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
2. Défago, X., Urbán, P., Hayashibara, N., Katayama, T.: Definition and specification of accrual failure detectors. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005), Yokohama, Japan, pp. 206–215 (2005)
3. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River (2005)
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
5. Ganesh, A.J., Kermarrec, A.-M., Massoulie, L.: Peer-to-peer membership management for gossip-based protocols. IEEE Transaction on Computers 52(2), 139–149 (2003)
6. Jayasinghe, D.: FAWS for SOAP-based web services (2005), http://www.ibm.com/developerworks/webservices/library/ws-faws/
7. Liang, D., Fang, C.-L., Chen, C., Lin, F.: Fault tolerant web service. In: Proc. of The 10th Asia-Pacific Software Engineering Conference (APSEC 2003), p. 310. IEEE Computer Society Press, Los Alamitos (2003)
8. Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the XOR metric. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 53–65. Springer, Heidelberg (2002)
9. Merideth, M.G., Iyengar, A., Mikalsen, T.A., Tai, S., Rouvellou, I., Narasimhan, P.: Thema: Byzantine-fault-tolerant middleware for web-service applications. In: SRDS, pp. 131–142. IEEE Computer Society Press, Los Alamitos (2005)
10. Reynal, M.: A short introduction to failure detectors for asynchronous distributed systems. SIGACT News (ACM Special Interest Group on Automata and Computability Theory) 36(1), 53–70 (2005)
11. Santos, G.T., Lung, L.C., Montez, C.: FTWeb: a fault tolerant infrastructure for web services. In: Proc. Ninth IEEE International EDOC Enterprise Computing Conference, pp. 95–105 (September 2005)
12. Zhao, W.: A lightweight fault tolerance framework for web services. In: Proc. IEEE/WIC/ACM International Conference on Web Intelligence, pp. 542–548 (November 2007)

# ReServE Service: An Approach to Increase Reliability in Service Oriented Systems*

Arkadiusz Danilecki, Mateusz Hołenko, Anna Kobusińska,
Michał Szychowiak, and Piotr Zierhoffer

Institute of Computing Science,
Poznań University of Technology, Poland
{Arkadiusz.Danilecki,Anna.Kobusinska,Michal.Szychowiak}@cs.put.poznan.pl,
{Mateusz.Holenko,Piotr.Zierhoffer}@gmail.com

**Abstract.** Heterogeneous environment of service-oriented computing turns to be very error-prone, due to the loose-coupling and a great number of independent components that are susceptible to failures. Therefore, we propose the ReServE service, which improves the reliability of the SOA-based systems and applications. The proposed service ensures that after a failure occurrence, the state of a business process is transparently recovered, and it is consistently perceived by the business process participants: clients and web services.

**Keywords:** service-oriented architectures, fault tolerance, rollback-recovery, web services.

## 1 Introduction

Nowadays, one of the major paradigms of the large scale distributed processing is service-oriented computing (SOA). According to the SOA reference model [13], in the service-oriented systems the applications are composed of loosely coupled, autonomous web services. Each service provides a well-defined, standardized interface that specifies how to use the service, and hides the details of the service implementation from the clients. Moreover, such a defined interface enables the cooperation among services, regardless of the diversity of the technology used, the geographical location and the organizational domains. By composing (also dynamically) the appropriate set of selected services, any required task is fulfilled, and a high degree of flexibility in the design of the SOA systems is achieved.

On the other hand, systems built according to the SOA paradigm inherit all the challenges associated with the construction of the more general distributed systems. In particular, the SOA systems are susceptible to faults, which are unavoidable in any large scale, distributed system (notably when it consists of many independent interacting components). Thus, providing the necessary level

* The research presented in this paper was partially supported by the European Union in the scope of the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 244–256, 2011.

of reliability of the SOA systems, which is maintained even in the case of failure of any system components, is needed.

There are many solutions of fault tolerance problem in general distributed systems [5,10]. Paradoxically, the features that determined the SOA attractiveness, such as a loose coupling, or high autonomy of the components – make direct usage of the known mechanisms difficult. For example, the classical solutions using the mechanisms of synchronous and asynchronous checkpointing [5] require either to control when the checkpoints are taken, or the appropriate choice of the checkpoint used during the process recovery. In the case of the SOA systems such solutions cannot be applied, because of the autonomy of the services. In addition, since every service invocation may result in irrevocable changes, it is necessary to apply so-called output-commit protocols [5], in which the pessimistic approach is used, and checkpoints are taken every time when the external interaction is performed.

Consequently, due to the specific characteristics of the SOA systems, the existing solutions providing fault-tolerance must be modified, and specially tailored for a service-oriented environments. Therefore, in this paper ReServE — the Reliability Service Environment, increasing the reliability of the SOA systems is presented. The proposed service, developed within the IT-SOA project [7], ensures that in the case of failure of one or more system components (i.e., web services or their clients), a coherent state of distributed processing is recovered. The ReServE service focuses on seeking automated mechanisms that neither require the user intervention in the case of failures, nor the knowledge of services' semantics. The proposed service provides also external support for services that do not have any mechanisms to ensure fault tolerance. While the ReServE service can be used in any SOA environment, it is particularly well-suited for the processing which does not have the transactional character, and for the applications that do not use the business process engines.

The rest of the paper is structured as follows: related work on fault tolerance in SOA systems is discussed in Section 2. Section 3 presents the system model and basic definitions. The architecture of the proposed ReServE service and its implementation details are described in Section 4. Section 5 presents the results of the simulation experiments. Finally, Section 6 concludes the paper.

## 2 Related Work

To improve reliability of the SOA-based systems, some solutions have been proposed [4,9]. A good example is the transaction processing. In the SOA systems it exists in different forms and requires different levels of isolation and atomicity [1]. In the transaction processing it is indispensable to have the possibility of rolling back the effects of the processing, in the case of failures of some performed operations. In such situations, the compensation of operations, realized in the SOA as the invocation of compensation services, is commonly used. A limitation of this approach is the necessity of providing all compensation services in advance, and the proper integration of the compensation invocations into processing, to ensure that the intended purpose of the rollback has been actually achieved.

Compensation mechanism can be also employed when transactions are rolled back for reasons not related to the failures of the system components (e.g. in the case of failures at the business logic level). Since the transactional approach [3,11] is burdened with high costs of maintenance transactions' properties, its use (and compensation) is not viable in applications that only require reliability.

Mechanisms improving reliability are to some extent implemented by many business processes engines (e.g. BPEL engines [8]). A common approach used by such engines is the forward recovery, mostly reduced to partially automatic retry of the failed operations. Business processes engines often provide the storage of the local processing state, which can be used to automatically recover, in the case of the engine failure and restart. Unfortunately, storing and recovering only the local processing state is not sufficient for the proper resumption of distributed processing.

The use of BPEL engines [8,14], and mechanisms they offer, cannot solve all the problems related to the issues of ensuring system reliability. Existing solutions increase the reliability of only a single component, which is a local instance of a business process implemented by the engine, without taking into account the potential dependencies between a nested services. As a result, such engines do not guarantee the preservation of exactly-once semantics for non-idempotent requests, unless additional protocols are employed (such as WS-ReliableMessaging [12]). They require the service developers' to prepare compensation procedures, and the business processes architects' to prepare the reactions to failures and to provide procedures for the exception handling (which requires the knowledge of the application logic and interactions semantics). Therefore, such solutions do not provide a fully automated and transparent recovery.

## 3 System Model and Basic Definitions

The services in the SOA-based systems are created and maintained by service providers ($SP$), and are used by service consumers ($SC$) i.e., the clients. A client requesting access to the service may not know in advance the identity of the $SP$, which will handle the request. The service may be *composite*, i.e. built of other services (some of which can be composite services as well). Moreover, the $SP$ may be unwilling to reveal the identities of the services it uses. As a result, the client may not know how many $SP$s are involved in the business process at any given time.

Business process is a set of logically related tasks that are performed to achieve business objectives. The definition of a business process specifies the behavior of its participants (clients and services) and describes the ordering of service invocations. Each business process consists of multiple interactions between $SC$s and $SPs$. During the interaction, an obligation (i.e. a promise of an action in the future) may be established. For example, a client booking cinema tickets is making an obligation (promise to pay). The server sending back the response is making obligation too (promise to reserve a place for the client). The obligation can be undone only explicitly with compensation procedures. If every obligation made during each interaction is kept (every party is ready for an action it

promised), the whole business process is consistent. Business process definition specifies when during an interaction an obligation is established. In our prototype we assume no information on business process definition, and in the consequence we assume that every message may transmit an obligation.

In the paper, we assume the crash-recovery model of failures [6] i.e. we do not tolerate byzantine failures and errors related to business logic. System components can fail at arbitrary moments, but every failure is eventually detected, for example by the failure detection service. The failed $SP$ becomes temporally unavailable until it is restored. The state of $SP$, which can be correctly reconstructed after a failure is called a *recovery point*. used.In order to create recovery points, logs and periodic checkpoints may be used according to the recovery policies choosen and pursued independently by each service provider. If SP employs checkpointing to implement its recovery policy, then we assume that $SP$ makes the decision to take a checkpoint independently, and in general, it may take no checkpoints at all. Similarly, we do not dictate the checkpoint policy to the $CP$. A recovery point may be also represented by a backup replica of the service. In particular, the initial $SP$ state constitutes the recovery point.

## 4 Reliability Service Environment

### 4.1 ReServE Service Architecture

The ReServE service architecture is presented in Fig. 1. The main module of the service is the *Recovery Management Unit* ($RMU$). Other two modules are proxy servers: *Client Proxy Unit* ($CPU$) and *Service Proxy Unit* ($SPU$). Their role is to hide the service architecture details from clients and services, respectively. The proxies are provided as a part of ReServE architecture and there is no need to implement a separate proxy for a new service. Any service at any moment may call other services; such a service becomes a client itself, and as a consequence, it has also its own $CPU$ apart from the $SPU$.

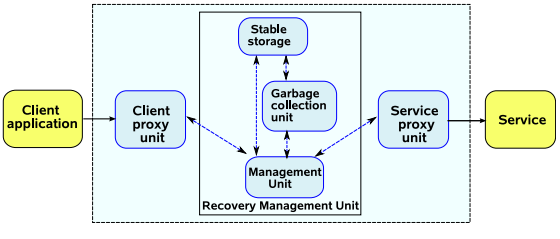



**Fig. 1.** ReServE service architecture

The $RMU$ records all invocations and responses sent between clients and services. In order to ensure the proper load balancing and high availability, the $RMU$ will be replicated. Three main modules of the $RMU$ are: *Stable Storage*, implemented over a database, *Management Unit* and *Garbage Collection Module*.

A *Garbage Collection Module* prevents the amount of data held by *Stable Storage* to grow indefinitely, by removing the information not used any longer. Because of the space limitations, in this paper we do not further explain this module's role.

In order to use the ReServE service, a client must use the *Client Proxy Unit*. *CPU* intercepts all requests issued by a client, and sends them to the *RMU*. The *CPU* may modify the client requests accordingly to the *RMU* requirements. In our prototype the *CPU* is a separate component located on the same machine as the client application; alternatively it may be a library, or even its functionality may be directly implemented by client application. The *CPU* must know the address of the *RMU*; it may be given by user or read from the configuration file. We assume that the *CPU* fails together with the client; the simultaneous failures are forced if necessary.

The *Service Proxy Unit* is located at the service provider site. It's primary task is monitoring the service and responding to service failures. In the case of failure (detected by the Failure Detection Service [2]), the *SPU* is responsible for initiating and managing the rollback-recovery process. The *SPU* serves a role of façade for the service: the service is available only via the *SPU*. There is exactly one *SPU* per service. Again, the *SPU* does not have to be a separate component; its functionality may be directly implemented by the service. The *SPU* has the addresses of the *RMU* and service written in the configuration file (alternatively, they may be given during the *SPU* startup), and enforces service registration in the *RMU*. Clients may use only registered services. We assume reliable communication link between the *SPU* and the service.

### 4.2 ReServE Service Requirements

In the SOA environment a service autonomy is one of the most important characteristics. Though there are many aspects of service autonomy, in the proposed solution we decided to concentrate on respecting the recovery policy autonomy. The *SP* may choose its own fault tolerance techniques (checkpointing, replication, logging) to implement service's high availability and reliability. The autonomy includes the parameters related to the technique (such as checkpoint frequency, number of replicas). *SP* may not be forced to take a checkpoint, create additional replicas, log messages it receives and sends. The *SP* may refuse to rollback the service to the previous recovery point.

In order to make the business process fault-tolerant some constraints on the autonomy of services must be put. The ReServE service is designed in the context of IT-SOA project and will be used by medical applications, which are still largely in the design phase. Therefore, we have the freedom to specify "reasonable" requirements for restrictions on service behavior, including both their interface and internal structure. We hope that they are minimal enough to make possible the integration of existing, legacy applications and services with ReServE service.

Both clients and services are expected to be piece-wise deterministic, i.e. they should generate the same results (in particular, the same URIs for a new

resource) in the result of multiple repetition of the same requests, assuming the same initial state. The client using ReServE service must have the unique identifier (`Client-Id`). This identifier may be given by a user when starting the client application. A client in order to utilize the proposed service has to use the *CPU* module of ReServE service, whose address it obtains from the configuration file. All requests should contain a unique sequence number (`Message-Id`), and the identifier of the performed business process (`Conversation-Id`). The `Message-Id` is necessary in order to distinguish the request retransmission due to failures from the intentional sending of two identical requests in a row. The `Message-Id`'s may be hardwired within the application code (as in our current prototype client applications). Initial values of both identifiers may be stored in a configuration file or in the *RMU* module. The services may call other services to fulfill client's request. The original client's `Conversation-Id` and `Client-Id` should be then attached to the invocations (in order to track dependencies). The *SP* should attach a unique sequence number `Response-Id` to the responses. The results of fulfilling the request may not be influenced by results of unfinished requests (similarly to an isolation property in transactions). The *SP*s may autonomously recover up to some point, according to its local recovery policy, e.g. with checkpointing and/or local logs. More than one recovery point may exist; the information on the recovery points must be available to the *RMU*. Each recovery point must be tagged with the identifier `Response-id` of the last served request. In addition, the recovery point should contain unacknowledged responses. If it is not possible then the service should allow *RMU* to request rollback to any recovery point (in order to recreate missing responses), and indicate when rollback has finished. While this violates one of aspects of *SP* autonomy, we consider this restriction acceptable for our target applications. Potentially *SP*s may tune their local recovery policy in order to balance runtime overhead vs. the recovery overhead. E.g. with checkpointing, the more often checkpoints are made by *SP*, the faster the system recovers. If the *SP*s checkpoint is invalid or damaged, we may use the earlier checkpoint and then roll forward using *RMU*s logs. In the extreme situation, there is a theoretical possibility of supporting services with no stable storage at all (though this is definitely *not* recommended).

### 4.3 Business Process Execution with the Use of the ReServE Service

**Failure-free processing.** When the user logs in to the client application it is asked for its `Client-Id`. The *RMU* may be then contacted to get the last saved client's state (which could be written during logout from the other machine). All requests first pass through the *CPU* (Fig. 2, step ①). The *CPU* augments the request when necessary (e.g. adding the client identifier `Client-Id`), and forwards the request to the *RMU* ②. The *RMU* maintains two queues in a Stable Storage. A `SavedRequests` queue indexed by `Client-Id` contains all requests received from clients. For each registered service there is a `SavedResponses` queue, containing accepted responses. The entries in the `SavedRequests` queue

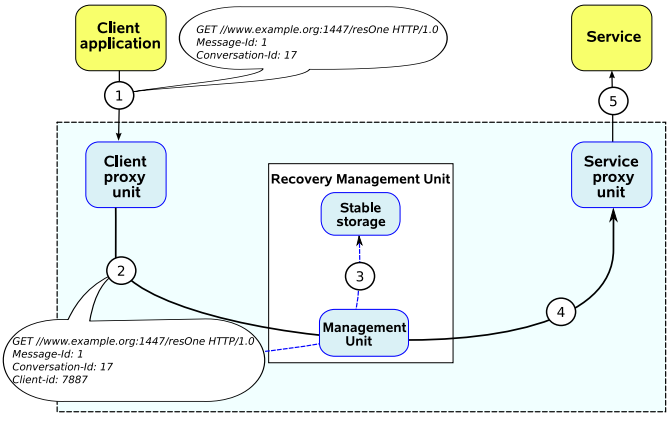



**Fig. 2.** Failure-free processing (1)

may contain reference to a relevant entry in the `SavedResponses`. Additionally, the *RMU* keeps in its volatile memory a queue of responses, which arrived from the service, but are not yet accepted.

After receiving the request, the *RMU* checks whether the Stable Storage already contains the response for a message with a given set of identifiers, and if so then then the response is send to the *CPU* and the request is ignored. The request is also ignored if it is contained in `SavedRequests` queue. Finally, if the request is not guaranteed to be read-only (e.g. the GET HTTP method) then the *RMU* saves it in the Stable Storage (3) in `SavedRequests` queue, and forwards the request to the *SPU* (4). The *SPU* forwards the request to the service (5). The service receives a request and executes it in accordance with its business logic. After the request's execution is completed, a response is generated and sent to the *SPU* (Fig. 3, step (1)). The `Response-Id` identifier attached to the response reflects the order in which the response was generated by the service. The *SPU* forwards response to the *RMU* (2).

After receiving the response, the *RMU* waits until all earlier responses from the service are received (all the response with smaller `Response-Id`). This forces the FIFO ordering of responses. When this condition is met, the response is accepted and logged in the stable storage (3) in the `SavedResponses` queue (even if this is a response for a read-only request). The reference to the response is added to appropriate entry in `SavedRequests` queue. Additionally, the response is also stored as the last response sent to the client who initiated the request, and the value `Response-Id` is stored as the `Last-Response` received from the service. Once this is done the response is passed to the *CPU* (4). The *CPU* removes all custom HTTP headers added to the response by the ReServE service components before passing it to the client (5).

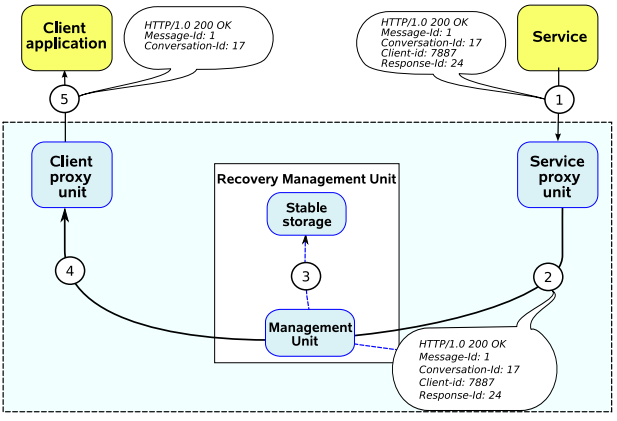



**Fig. 3.** Failure-free processing (2)

In order to tolerate transient communication links faults etc, the messages may be retransmitted periodically; the duplicates are detected by the *RMU*, the *SPU* and the *CPU* using set of message identifiers. Therefore, *exactly once* delivery is guaranteed in the case of failure-free execution. The user may switch between different machines at any moment; the user may note at the logout that the state of a client application should be sent to the *RMU*. During logging in to the new machine, the user may request the old state from the *RMU*, identified by the `Client-Id`. The security of the data currently is protected by simple password, however the the stronger protection will be needed in the future.

**Client Application Failure.** If the client crashes, the user may start recovery by starting the client application again (possibly on the different machine than previously) and giving its `Client-Id`. The recovery of the client differs depending on the client's requirements. For some clients, the last response from the service may be enough for recovery. Such clients may contact the *RMU*, request the last response stored by the *RMU*, and then directly proceed with the execution. If the last response is not sufficient for client's revival, it must cooperate with ReServE service. At the beginning, the client should first recover using its own local checkpoints or logs. Clients without stable storage may use the *RMU* as a remote storage service. Next the client proceeds with processing, sending requests to the *CPU*, which then forwards them to the *RMU*. If the *RMU* already has the response for the request, such a response is sent to the client. Since the client is piece-wise deterministic, it should reconstruct its state up to the point of the last request sent before the failure.

Finally the client may sent a request for which the *RMU* has no response stored. This request is ignored if it was already exists in the `SavedRequests` queue (since it means the request was already received in the past and transmitted to the proper *SPU*). Otherwise, the *RMU* forwards the request to the appropriate *SPU*; the client's recovery is finished at this point.

**Service Failure.** Once a service's failure is detected by the *SPU*, or if the *SPU* fails itself, the service's rollback-recovery process starts. First, the *SPU* asks the service for the list of available checkpoints, along with the `Saved-Response` identifier. If the last available recovery point contains responses, they are first sent to the *RMU*, and the *SPU* waits until they are acknowledged by the *RMU* before proceeding with recovery. Afterwards, the *SPU* asks the *RMU* for the value of the `Last-Response`

Since some client may already received a response reflecting some service state $S_m$ (its identifier is given by the `Last-Response`). If during a local recovery service is rolled back to some earlier state $S_n$, we must ensure that the state $S_m$ is recovered in order to ensure the consistency between the client's and the service's state. The state $S_m$ was a result of executing some sequence of requests starting with state $S_n$. Since we assume the service is piece-wise deterministic, starting with the state $S_n$ and executing the same sequence of requests during the recovery must in effect recreate lost $S_m$ state. For each request, we know its the ordering thanks to the `Response-Id` attached to the response. The problem is only finding proper starting point of re-execution.

If the `Saved-Response` is greater than the `Last-Response`, then it means that before the recovery point the service has executed several request for which we lost the ordering information. Therefore, the service must be rolled back to the latest checkpoint for which the `Saved-Response` is less from or equal to the received `Last-Response`. After the rollback is completed, the *SPU* asks the *RMU* for a sequence of requests, attaching the `Saved-Response` of the chosen recovery point to the request. The *RMU* selects from its stable storage all requests with no response saved (i.e. with no ordering information) or for which the response contains the identifier `Response-Id` greater than or equal to the `Saved-Response` value received from the *SPU*. In order to inform the *SPU* on the original order of request execution, the `Response-Id` is attached to the request, but only if the *RMU* has a response for a request and its `Response-Id` is less than `Last-Response`. The responses with `Response-Id` greater than `Last-Response` are purged from the logs. After receiving requests from the *RMU*, the *SPU* may start the recovery. First the request for which ordering is known are passed sequentially to the service, in the order corresponding with their `Response-Id`. Each time the *SPU* waits for response before sending next request. Then the *SPU* finishes the recovery and passes the remaining requests for execution, in any order. Any error during recovery causes the *SPU* to signal a recovery failure. It's up to client or transaction protocols to further treat the situation.

In the case when the service has no stable storage and has not used the ReServE service as a remote storage, it will be rolled back to the initial state, and *all* client's requests will be repeated, in order to obtain the lost service state. Since executing some requests may involve interaction with an outside world, our solution needs to guarantee that some service states are *never* forgot; they may be rolled back, but only if re-execution results in exactly the same service's state. In our prototype we require any interaction with the outside world to

be modeled as a call to an external service. The service should guarantee, that during re-execution interactions with the outside world are not repeated.

A failure of one service should not force other services to rollback. In addition, the forced rollback of the service should not cause the rollback of other services. In our solution, if the service state reflects sending some responses while those responses are lost, we force service to earlier recovery point in order to regenerate some responses. This rollback may then force also cascading rollback of the services invoked after the recovery point. This would however be major violation of the *SP* autonomy and therefore, currently we stall the calls to external services whenever there is possibility that our architecture would be unable to achieve the consistent state of a business process.

## 5 Simulation Experiments and Performance Evaluation

The performance of the proposed ReServE service is quantitatively evaluated in terms of the overhead introduced by the service in the case of the failure-free processing. In order to guarantee the correct system behavior in the case of failure occurrence, the significant performance loss is to be expected. Therefore, the simulation experiments were performed, to estimate the order of magnitude of the overhead introduced by the ReServE service, and to examine, whether the performance loss connected with the use of the proposed service is acceptable, or it is unacceptably high. Moreover, the impact of the message size on the introduced overhead was investigated, to check how the characteristics of the overhead change within a range of possible message sizes expected in the target applications. Finally, the location of the RMU module on the recovery time was assessed.

In the performed experiments, the workstations with the following characteristics: SuSE Linux 11.1 kernel 2.6.27.45-0.1-default operating system, with Intel Pentium 4 3.20GHz x 2 processor, and 2.9 GB RAM were used. The service and its proxy server run on the same workstation — the service is implemented with the RestLet 1.1. environment, while its proxy server uses proxy server MProxy 0.4 [2]. Stable Storage is developed as a PostreSQL database. As a client application and its proxy server the JMeter 2.3.4 software was used.

The simulation experiments were performed, successively for 10 to 100 threads per client workstation. Since ten workstations were used as clients, up to 1000 threads were running. Each thread repeated the simulation experiments 500 times. It was assumed that the time associated with the realization of the service is constant, and is set to 100 ms (as for our target applications, in general the service's request processing time depends neither on the number of clients, nor on the size of data), which was assumed to be the highest range of the expected performance time of the web applications developed for the IT-SOA project.

The overhead introduced by the ReServE service in the case of the failure-free processing encompasses the time associated with the communication, service processing, saving data in the Stable Storage of RMU, and finally interaction with a client. The overhead is measured at the client application side, and is

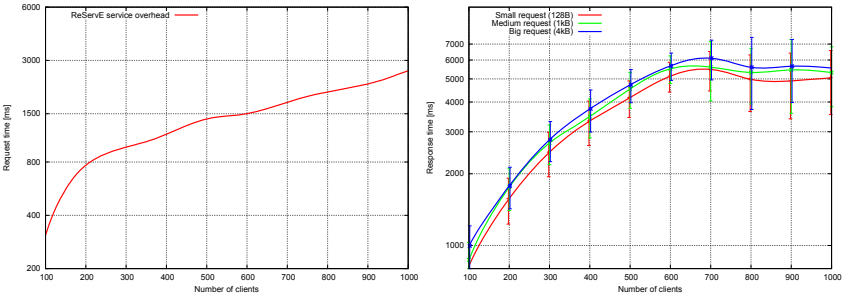



**Fig. 4.** (a) ReServE service overhead (b) The impact of the message size on the overhead introduced by the ReServE service

obtained by calculating the difference between processing a service with the use of the ReServE service and without it. The obtained results are presented in the Fig. 4(a). The overhead introduced by the ReServE service is largely connected with saving data in the Stable Storage of the RMU. Such data are written on the hard drive, using the PostgreSQL database. In order to reduce the obtained overhead several steps can be taken. Fist, the used PostgreSQL databased should be properly tuned. Also, as an alternative to writing data to the hard drive, the in-memory logging techniques known from message-passing systems could be applied [5]. Finally, since the single RMU module can constitute the system bottleneck, the distribution of this module could increase the efficiency of the ReServE service, and thus decrease the introduced overhead. For some clients the obtained overhead could be unacceptable. However, since the ReServE service within the IT-SOA project is intended primarily for the use with interactive services, where a client enters the data to the service and such an operation takes some time (e.g. ReServE service client is a doctor examining a patient and placing information on his health state), then in the case of such a client, the overhead introduced by the ReServE service seems to be relatively small. For clients using interacting services the proposed solution seems to be attractive, as the overhead introduced by the ReServE service is acceptable, and simultaneously the clients obtain the guarantee of no data loss.

In the Fig. 4(b) the impact of the size of sent messages on the overhead introduced by the ReServE service is shown. In the performed experiments three types of messages were considered: 128B, 1 kB, and 4kB. Based on the values presented in the Fig. 4(b), it is clear, that while the message size has small, but noticeable influence on the overhead introduced by the proposed service, it does not change the overhead's characteristics. Such an overhead is composed of three factors: the time of the message transmission, the delay associated with saving it to the Stable Storage, and the impact of increased service demands by the libraries of individual clients. It can be observed that the cost of servicing a large demand is greater for a large number of customers. This result is actually expected, as with the higher number of clients, the higher system load and the increased consumption of resources occurs.

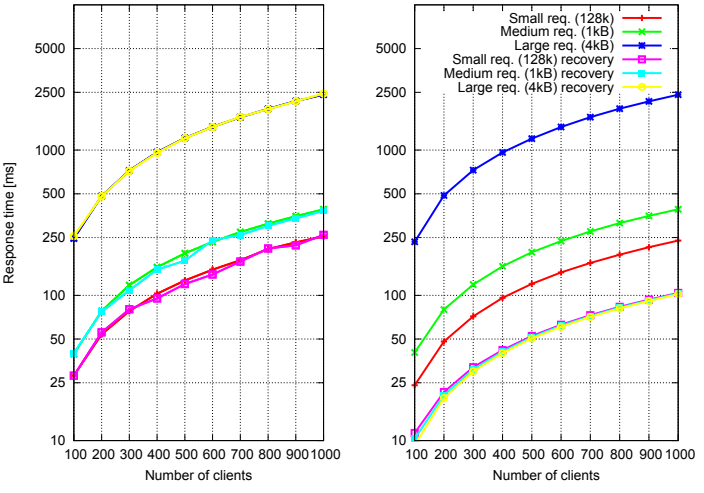



**Fig. 5.** Impact of the RMU location on recovery time

Further experiments are related to the analysis of RMU module location impact on the recovery time in the case of failure occurrence. The obtained results are presented in the Fig. 5. The tests were performed for two configurations of the environment: the RMU located in the local network of the client application, and at the service side. According to the obtained results, when the RMU is placed in the local network with the web services, the recovery time is relatively short. This follows from the high bandwidth of the local network communication links.

## 6 Conclusions and the Future Work

This paper describes the ReServE service providing support to the recovery of web services. The proposed service respects web services local recovery autonomy, and do not force any particular technique to create service recovery points. It allows to recreate lost service states in the case, when the local recovery policy is unable to achieve this (e.g. with damaged checkpoint files, or obsolete service replicas). For the moment being, the prototype of the proposed service is under constant improvement: currently, the RMU neither requires, nor needs any preliminary knowledge on the service structure, business process definition, or application logic (e.g. which messages transmit obligations); many optimizations are possible if the RMU would have an access to such an information. The future work on ReServE service assumes further improvement of the proposed service efficiency and the distribution of the the *RMU* module.

## References


1. Adya, A., Liskov, B., O'Neil, P.E.: Generalized Isolation Level Definitions. In: ICDE, pp. 67–78. IEEE Computer Society, Los Alamitos (2000)
2. Brzeziński, J., Kalewski, M., Libuda, M., Sobaniec, C., Wawrzyniak, D., Wojciechowski, P., Wysocki, P.: Replication Tools for Service-Oriented Architecture. In: SOA Infrastructure Tools— Concepts and Methods, pp. 153–179. Pozna University of Economics Press (2010)
3. Cabrera, L.F., Copeland, G., Cox, B., Freund, T., Klein, J., Storey, T., Thatte, S.: Web services transactions specifications (2005), http://www-106.ibm.com/developerworks/-webservices/library/ws-transpec/
4. Chan, P.P.W., Lyu, M.R., Malek, M.: Reliable Web Services: Methodology, Experiment and Modeling. In: ICWS, pp. 679–686. IEEE Computer Society, Los Alamitos (2007)
5. Elnozahy, E.N., Lorenzo, A., Wang, Y.M., Johnson, D.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys 34(3), 375–408 (2002)
6. Guerraoui, R., Rodrigues, L.: Introduction to Distributed Algorithms. Springer, Heidelberg (2004)
7. IT-SOA consortium: IT-SOA project (2009), http://www.soa.edu.pl/
8. Lau, J., Lung, L.C., da Silva Fraga, J., Veronese, G.S.: Designing Fault Tolerant Web Services Using BPEL. In: Lee, R.Y. (ed.) ACIS-ICIS, pp. 618–623. IEEE Computer Society, Los Alamitos (2008)
9. Maamar, Z., Sheng, Q.Z., Tata, S., Benslimane, D., Sellami, M.: Towards an Approach to Sustain Web Services High-Availability Using Communities of Web Services. International Journal of Web Information System 5(1), 32–55 (2009)
10. Maloney, A., Goscinski, A.: A Survey and Review of the Current State of Rollback-Recovery for Cluster Systems. Concurrency and Computation: Practice & Experience 21(12), 1632–1666 (2009)
11. Marinos, A., Razavi, A.R., Moschoyiannis, S., Krause, P.J.: RETRO: A Consistent and Recoverable RESTful Transaction Model. In: ICWS, pp. 181–188. IEEE, Los Alamitos (2009)
12. OASIS: Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.1 (January 2008), http://docs.oasis-open.org/ws-rx/wsrm/v1.1/wsrm.html
13. OASIS: Reference Architecture Foundation for Service Oriented Architecture - Version 1.0 (October 2009)
14. Oracle Corporation: Documentation for Oracle BPEL Process Manager and Human Wokflow (2009), http://www.oracle.com/-technology/-products/-soa/bpel/-collateral/documentation.html

# Hypergraph Partitioning for the Parallel Computation of Continuous Petri Nets*

Zuohua Ding[1], Hui Shen[1], and Jianwen Cao[2]

[1] Center of Math Computing and Software Engineering,
Zhejiang Sci-Tech University,
Hangzhou, Zhejiang, 310018, China
[2] Institute of Software,
Chinese Academy Sciences,
Beijing, 100080, China

**Abstract.** Continuous Petri net can be used for performance analysis or static analysis. The analysis is based on solving the associated ordinary differential equations. However, large equation groups will give us overhead computing. To solve this issue, this paper presents a method to compute these differential equations in parallel. We first map the Petri net to a hypergraph, and then partition the hypergraph with minimal inter-processor communication and good load balance; Based on the partition result, we divide the differential equations into several blocks; Finally we design parallel computing algorithm to compute these equations. Software hMETIS and SUNDIALS have been used to partition the hypergraph and to support the parallel computing, respectively. Gas Station problem and Dining Philosopher problem have been used to demonstrate the benefit of our method.

**Keywords:** Continuous Petri net, ODE, hypergraph, parallel computing.

## 1 Introduction

In order to alleviate or avoid state explosion problem, continuous Petri net(CPN) can be used for performance analysis[11][19] or static analysis[8] for some parallel/distributed systems. There are two kinds of Continuous Petri net based on the firing style: one is that the instantaneous firing speed of a transition is proportional to the minimum of the markings of the input places, and the other is that the instantaneous firing speed of a transition is proportional to the product of the markings of the input places. We call the first one Minimum-CPN and the second one Product-CPN.

For Minimum-CPN[6], since the marking values of places equal to the expected values of places in the general stochastic Petri net[11], it is proper to be used for performance analysis. For Product-CPN, since the firing rates magnify the states, and also the marking at each place is continuous without points of discontinuity, it can be used to reveal some extreme behavior such as system deadlock without hitting state explosion

* This work is supported by the NSF under Grant No. 90818013 and Zhejiang NSF under Grant No.Z1090357.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 257–271, 2011.

problem[8]. No matter which kind of CPN, the analysis is always based on solving the associated ordinary differential equations.

If a system is very large, then the number of ordinary differential equations could be very big, or the number of differential equation groups could be very big. For example, to analyze a system with $10^{20}$ states that has been used for Symbolic Model Checking by Burch et al.[2], we need to solve an equation group with $10^{20}$ equations. To solve such large equation group, the normal equation solver such as Matlab may not have enough power to perform such computing. A solution is to solve the equation group in parallel.

In this paper, we present a new method for the parallel computing of the ordinary differential equations. The following is the sketch of the process. We first map the Petri net to a hypergraph, and then partition the hypergraph by hMETIS software. Based on the partition result, we divide the differential equations into several blocks. Finally we design parallel computing algorithm to compute these equations. PVODE package from SUNDIALS is used to parallel compute the differential equations.

Even through our method is developed for Product-CPN, it can also be applied to Minimum-CPN.

This paper is organized as the following. Section 2 describes the Ordinary Differential Equation (ODE) representation of concurrent programs. Section 3 shows how to compute the ODE group in parallel. Section 4 is the case study. Gas station problem has been used to illustrate how to apply our method. Section 5 is also a case study. More experiments have been conducted for dining philosopher problem. Section 6 is a discussion of the related work. The last section, Section 7, is the conclusion of the paper.

## 2 ODE Representation of Concurrent Systems

In this section, we briefly describe how to use ordinary differential equations to represent a concurrent system. For the details, please see our previous work[8]. We consider a subclass of Petri nets, namely MP net.

**Definition 1.** *A MP net is a tuple* $< P, T >$*, where*

1. $P = \{p_1, p_2, ..., p_n\}$ *is a finite nonempty set of places,*
2. $T = \{t_1, t_2, ..., t_m\}$ *is a finite nonempty set of transitions.*

*Meanwhile, the following conditions are satisfied:*

- *the places of P are partitioned into three disjoint partitions I, B and C;*
- *each place from I has at least one input transition and at least one output transition;*
- *each place from B has one input transition and at least one output transition or has one output transition and at least one input transition;*
- *each place from C has at least one pair of input transition and output transition;*
- *each transition has one input place from I and one output place from I; and*
- *each transition has either*
  - *no input places from B/C and no output places from B/C (*internal transition*); or*

- *one input place from B and one output place from C(input communication transition); or*
- *one output place from B and one input place from C(output communication transition).*

*Here I stands for Internal Places, B stands for Buffer and C stands for Control.*

**Definition 2.** *A Continuous Petri Net is a tuple* $CPN =< PN^{\mathcal{M}}, M_0, C >$*, where*

1. $< PN^{\mathcal{M}}, M_0 >$ *is a marked MP net, where* $M_0$ *is the initial marking.*
2. $C : T \to (0, +\infty)$, $C(t_j) = c_j\ (j = 1, 2, \ldots, m)$ *is a mapping to assign a firing constant* $c_j$ *to transition* $t_j$*.*

**Definition 3.** *Let* $I = [0, +\infty)$ *be the time interval and let* $m_i : I \to [0, +\infty)$, $i = 1, 2, \ldots, n$ *be a set of mappings that associated with place* $p_i$*. A marking of a Continuous Petri Net* $CPN =< PN^{\mathcal{M}}, M_0, D >$ *is a mapping*

$$m : I \to [0, +\infty)^n,\ m(\tau) = (m_1(\tau), m_2(\tau), \ldots, m_n(\tau)).$$

The follows are the semantics of our continuous Petri net, which show how to calculate the markings of places. We have the following cases. Let $p$ denote place, $t$ denote transition, $m$ denote marking, and $c$ denote firing constant.

*Case 1*. No inputs for the net as Fig. 1 shows.

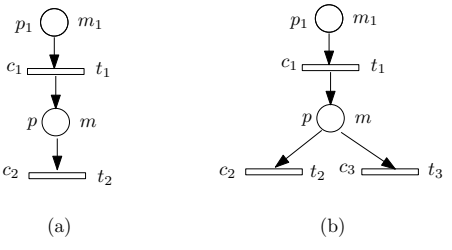



**Fig. 1.** Net without inputs

(a) Place has no choice. For the net in Fig. 1(a), the differential equation is:

$$m'(\tau) = c_1 m_1(\tau) - c_2 m(\tau).$$

(b) Place has choice. For the net in Fig. 1(b), the differential equation is:

$$m'(\tau) = c_1 m_1(\tau) - (c_2 + d_3) m(\tau).$$

*Case 2*. One input for the net as Fig. 2 shows.

(a) Input Before. For the net in Fig. 2(a), the differential equation is:

$$m'(\tau) = c_1 m_1(\tau) m_2(\tau) - c_2 m(\tau).$$

(b) input After. For the net in Fig. 2(b), the differential equation is:

$$m'(\tau) = c_1 m_1(\tau) - c_2 m(\tau) m_2(\tau).$$

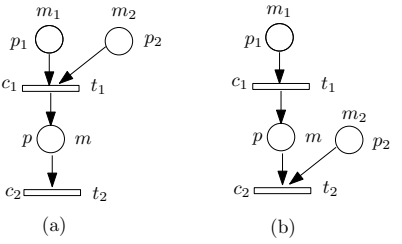



**Fig. 2.** Net with one input

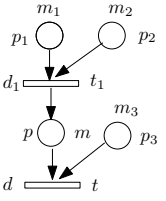



**Fig. 3.** Net with two inputs

*Case 3*. Two inputs for the net. For the net in Fig. 3. The differential equation is:

$$m'(\tau) = c_1 m_1(\tau) m_2(\tau) - c_2 m(\tau) m_3(\tau).$$

Eventually, we will get four types of differential equations as the follows($x_i$ represent the variables):

- Type 1. $x_i' = c_{i-1} x_{i-1} - c_i x_i$.
- Type 2. $x_i' = c_{i-1} x_{i-1} x_k - c_i x_i$.
- Type 3. $x_i' = c_{i-1} x_{i-1} - c_i x_i x_k$.
- Type 4. $x_i' = c_{i-1} x_{i-1} x_k - c_i x_i x_l$.

Hence, the markings of places of the continuous Petri net can be solved from a set of ordinary differential equations consisting of above four types.

## 3 Parallel Computing of Differential Equations

We choose hypergraph for parallel computing because hypergraph models can accurately represent communication volume [5]. In parallel computing, communication is required for a hyperedge whose vertices are in two or more processors. Çatalyürek and Aykanat [5] proposed a hypergraph model for sparse matrix-vector multiplication, and showed that the hyperedge cut metric corresponds exactly to the communication volume. An important advantage of the hypergraph model is that it can easily represent nonsymmetric and rectangular matrices.

### 3.1 Moving from Petri Net To Hypergraph

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. Every net $n_j \in \mathcal{N}$ is a subset of vertices, i.e., $n_j \subset \mathcal{V}$. The vertices in a net are called its *pins*. Weights can be associated with the vertices of a hypergraph. Let $w_i$ denote the weight of vertex $v_i \in \mathcal{V}$.

Let $\mathcal{P}$ be a MP that has $n$ process cycles: $P_1, P_2, \ldots, P_n$. $m_1, m_2, \ldots, m_K$ are the buffers(or Resources) for these process cycles. The corresponding hypergraph of $\mathcal{P}$ is $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, which can be obtained based on the following translation rules:

1) Each $P_i$ corresponds to a vertex $v_i$ of $\mathcal{H}$. If $P_i$ has $n_i$ places, then the weight of $v_i$ is $w_{v_i} = n_i$;

2) Each $m_i$ corresponds to a vertex $v'_i$ of $\mathcal{H}$, the weight of $v'_i$ is $w_{v'_i} = 1$;

3) For any two process cycles $P_i, P_j$, if they have buffers $m_{i_1}, \ldots, m_{i_k}$, then $\{P_i, P_j, m_{i_1}, \ldots, m_{i_k}\}$ corresponds to an edge $e$ of $\mathcal{H}$, denoted as $e = \{v_i, v_j, v'_{i_1}, \ldots, v'_{i_k}\}$. If there exist total $n_{ij}$ directed arcs between $m_{i_1}, \ldots, m_{i_k}$ and $P_i, P_j$, then the weight of $e$ is $w_e = n_{ij}$.

Particularly, based on these rules, we can obtain hypergraphs for those three synchronization structures, as shown in Fig. 4, Fig.5, Fig.6.

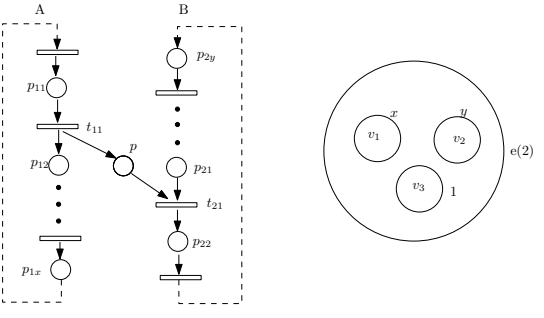



**Fig. 4.** Hypergraph for asynchronous message passing

**Remark 4.** *The following factors are considered when we define the translation rules:*

- *We did not consider the directions of the hypergraph when translated from directed Petri net since the partition of hypergraph is based on the undirected graph.*
- *Since the places in the same process cycles are dependent to each other, we map those places to one vertex in the hypergraph; otherwise, if we map them to different vertices, it is possible that these vertices may fall to different parts after partition of hypergraph, which may cause the communication times to increase.*
- *We map the process cycles and the related communication places to an edge since the edge should contain as more dependent vertices as possible[13].*
- *The weight of vertex is based on the number of places, which indicates the size of the task and the required processing time; The weight of the edge is based on the number of arcs that connect buffers(resources) to the process cycles, which represents the data dependent relation and the communication time.*

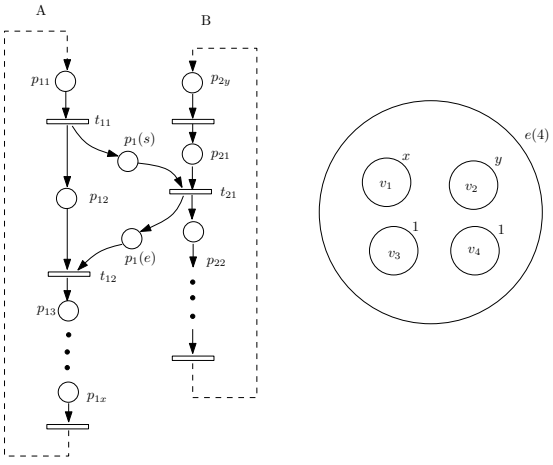



**Fig. 5.** Hypergraph for synchronous message passing (I)

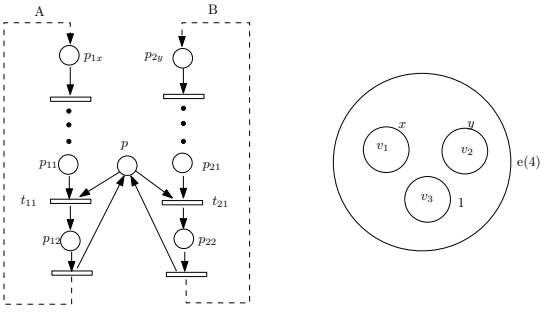



**Fig. 6.** Hypergraph for resource sharing

### 3.2 Partitioning Hypergraph

After we get hypergraph from Petri net, we need to partition this hypergraph. In general we need to compute a K-way partition [4].

A K-way vertex partition $\mathcal{K} = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ of $\mathcal{H}$ is said to be balanced if each part $\mathcal{V}_k$ satisfies the balance criterion

$$W_k \leq W_{avg}(1+\epsilon), \quad \text{for } k = 1, 2, \ldots, K.$$

Here weight $W_k = \sum_{v_i \in \mathcal{V}_k} w_i$, $W_{avg} = (\sum_{v_i \in \mathcal{V}} w_i)/K$, and $\epsilon$ represents the predetermined maximum imbalance ratio allowed.

In a partition $\mathcal{K}$ of $\mathcal{H}$, a net that has at least one pin in a part is said to connect that part. *Connectivity set* $\Lambda_j$ of a net $n_j$ is defined as the set of parts connected by $n_j$. *Connectivity* $\lambda_j = |\Lambda_j|$ of a net $n_j$ denotes the number of parts connected by $n_j$. A net $n_j$ is said to be *cut* if it contains more than one part (i.e. $\lambda_j > 1$), and *uncut* otherwise

(i.e. $\lambda_j = 1$). The set of cut nets of a partition $\mathcal{K}$ is denoted as $\mathcal{N}_{\mathcal{E}}$. The cutsize is defined as

$$cutsize(\mathcal{K}) = \sum_{n_j \in \mathcal{N}_{\mathcal{E}}} (\lambda_j - 1).$$

Hence, the hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion among the part weights is maintained. The hypergraph partitioning problem is known to be NP-hard [17].

hMETIS is a software package for partitioning large hypergraphs, especially those arising in circuit design. The algorithms in hMETIS are based on multilevel hypergraph partitioning described in [16].

### 3.3 Parallel Computing

Based on the partitioning result of the hypergraph, we can divide the ordinary differential equations into several small groups. Our parallelization is based on parallelism in space since our equation system has a regular structure.

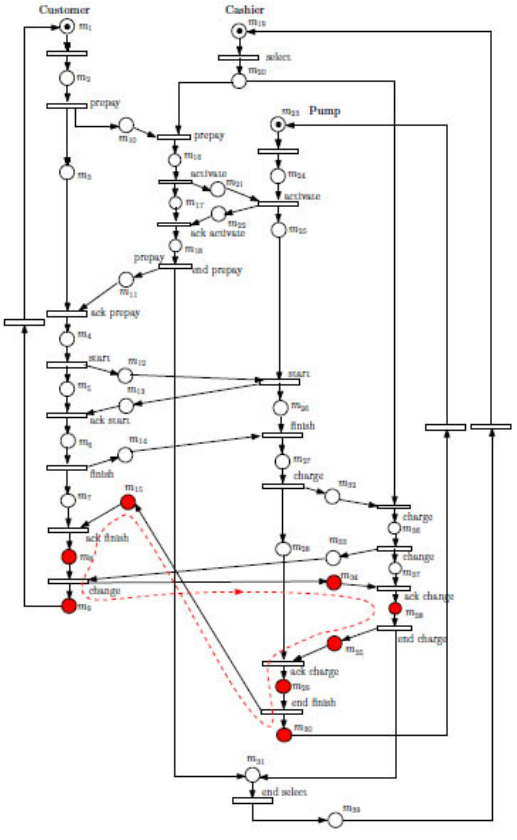
Software SUNDIALS [14] is used to support our parallel computing. One of its package, called *PVODE* is used to solve our differential equations in parallel. *PVODE* can solve stiff and nonstiff initial value problems. In our case, our equation group will display some stiffness if we have some big $d_i$ and small $d_i$ in the equation group.

Besides communication part, a critical part of PVODE, that makes it an ODE parallel "solver" rather than just an ODE parallel method, is its control of local error. At every step, the local error is estimated and required to satisfy tolerance conditions, and the step is redone with reduced step size whenever that error test fails. PVODE includes an algorithm, STALD(STAbility Limit Detection), which provides protection against potentially unstable behavior of the BDF multistep integration methods in certain situations. Thus the computing error will not affect our performance analysis.

## 4 Case Study (I): The Gas Station Program

The Gas-Station problem which models activities of an automated gas station was originally described in [12]. This example has been widely studied for property analysis, specially deadlock analysis. Generally, the automated gas station consists of a set of cashiers, a set of pumps and a set of customers. The scenario is as follows: customers arrive and pay the casher for gas. The cashier activates a pump, at which the customer then pumps gas. When the customer is finished, the pump reports the amount of gas actually pumped to the cashier, who then gives the customer the change. The Petri net representation and the corresponding ordinary differential equation model are shown in Fig. 7. The net has three process cycles: $P_1$ for customer, $P_2$ for cashier, and $P_3$ for pump.

The initial values for this equation group are: $m_1(0) = 1$, $m_{19}(0) = 1$, $m_{23}(0) = 1$, and all others are 0.




$$
\begin{cases}
m'_1 = m_9 - m_1 \\
m'_2 = m_1 - m_2 \\
m'_3 = m_2 - m_3 \times m_{11} \\
m'_4 = m_3 \times m_{11} - m_4 \\
m'_5 = m_4 - m_5 \times m_{13} \\
m'_6 = m_5 \times m_{13} - m_6 \\
m'_7 = m_6 - m_7 \times m_{15} \\
m'_8 = m_7 \times m_{15} - m_8 \times m_{33} \\
m'_9 = m_8 \times m_{33} - m_9 \\
m'_{10} = m_2 - m_{10} \times m_{20} \\
m'_{11} = m_{18} - m_3 \times m_{11} \\
m'_{12} = m_4 - m_{12} \times m_{25} \\
m'_{13} = m_{12} \times m_{25} - m_5 \times m_{13} \\
m'_{14} = m_6 - m_{14} \times m_{26} \\
m'_{15} = m_{29} - m_7 \times m_{15} \\
m'_{16} = m_{10} \times m_{20} - m_{16} \\
m'_{17} = m_{16} - m_{17} \times m_{22} \\
m'_{18} = m_{17} \times m_{22} - m_{18} \\
m'_{19} = m_{39} - m_{19} \\
m'_{20} = m_{19} - m_{10} \times m_{20} - m_{20} \times m_{32} \\
m'_{21} = m_{16} - m_{21} \times m_{24} \\
m'_{22} = m_{21} \times m_{24} - m_{17} \times m_{22} \\
m'_{23} = m_{30} - m_{23} \\
m'_{24} = m_{23} - m_{21} \times m_{24} \\
m'_{25} = m_{21} \times m_{24} - m_{14} \times m_{25} \\
m'_{26} = m_{12} \times m_{25} - m_{14} \times m_{26} \\
m'_{27} = m_{14} \times m_{26} - m_{27} \\
m'_{28} = m_{27} - m_{28} \times m_{35} \\
m'_{29} = m_{28} \times m_{35} - m_{29} \\
m'_{30} = m_{29} - m_{30} \\
m'_{31} = m_{18} + m_{38} - m_{31} \\
m'_{32} = m_{27} - m_{20} \times m_{32} \\
m'_{33} = m_{36} - m_8 \times m_{33} \\
m'_{34} = m_8 \times m_{33} - m_{34} \times m_{37} \\
m'_{35} = m_{38} - m_{28} \times m_{35} \\
m'_{36} = m_{20} \times m_{32} - m_{36} \\
m'_{37} = m_{36} - m_{34} \times m_{37} \\
m'_{38} = m_{34} \times m_{37} - m_{38} \\
m'_{39} = m_{31} - m_{39}
\end{cases}
$$

**Fig. 7.** Petri net model and equation model for gas station problem

### 4.1 From Petri Net to Hypergraph

Following our translation rules, we get: $P_1 \rightarrow v_1$, $P_2 \rightarrow v_2$, $P_3 \rightarrow v_3$, $m_{10} \rightarrow v_4$, $m_{11} \rightarrow v_5$, $m_{12} \rightarrow v_6$, $m_{13} \rightarrow v_7$, $m_{14} \rightarrow v_8$, $m_{15} \rightarrow v_9$, $m_{21} \rightarrow v_{10}$, $m_{22} \rightarrow v_{11}$, $m_{32} \rightarrow v_{12}$, $m_{33} \rightarrow v_{13}$, $m_{34} \rightarrow v_{14}$, $m_{35} \rightarrow v_{15}$, $e = \{v_1, v_2, v_4, v_5, v_{13}, v_{14}\}$, $e_2 = \{v_2, v_3, v_{10}, v_{11}, v_{12}, v_{15}\}$, $e_3 = \{v_1, v_3, v_6, v_7, v_8, v_9\}$, $w_{v_1} = 9$, $w_{v_2} = 10$, $w_{v_3} = 8$, $w_{v_4} = w_{v_5} = w_{v_6} = w_{v_7} = w_{v_8} = w_{v_9} = w_{v_{10}} = w_{v_{11}} = w_{v_{12}} = w_{v_{13}} = w_{v_{14}} = w_{v_{15}} = 1$, $w_{e_1} = w_{e_2} = w_{e_3} = 8$. The corresponding hypergraph is shown in Fig. 8(a).

### 4.2 Partition of the Hypergraph of Gas Station

By using software hMETIS, we can partition the hypergraph of gas station problem. The parameters have been designed as the following. $Nparts = 3$, indicating the graph will be partitioned to 3 parts; $UBfactor = 3$, specifying the allowed imbalance between the partitions during recursive bisection; $Nruns = 10$, indicating the number of the

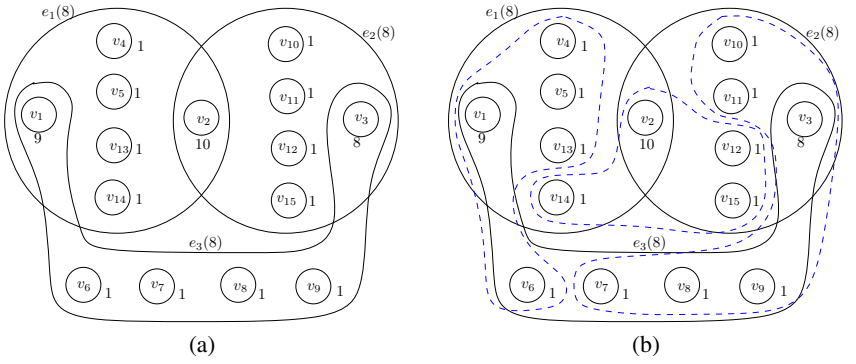



**Fig. 8.** (a) Hypergraph for gas station problem. (b) Partition of hypergraph for gas station problem.

different bisections that are performed by hMETIS; $CType = 1$, indicating that during the coarsening phase, hybrid first-choice scheme (HFC) has been used as the vertex grouping scheme; $RType = 1$, indicating that during the uncoarsening phase, Fiduccia-Mattheyses (FM) refinement scheme has been used; $Vcycle = 3$, indicating that each one of the Nruns bisections is refined using V-cycles; $Reconst = 1$, meaning that the chosen scheme reconstructs the hyperedges that are being cut; $dbglvl = 24$, meaning to request hMETIS to print debugging information.

The resulting partition is as the following: $v_2$, $v_{12}$, $v_{14}$, $v_{15}$ are in the first part, $v_3$, $v_7$, $v_8$, $v_9$, $v_{10}$, $v_{11}$ are in the second part, and $v_1$, $v_4$, $v_5$, $v_6$, $v_{13}$ are in the third part. The weights for these three parts are:

$$w_1 = w_{v_2} + w_{v_{12}} + w_{v_{14}} + w_{v_{15}} = 10 + 1 + 1 + 1 = 13,$$
$$w_2 = w_{v_7} + w_{v_8} + w_{v_9} + w_{v_{10}} + w_{v_{11}} + w_{v_3} = 1 + 1 + 1 + 1 + 1 + 8 = 13,$$
$$w_3 = w_{v_1} + w_{v_4} + w_{v_5} + w_{v_{13}} + w_{v_6} = 9 + 1 + 1 + 1 + 1 = 13.$$

Since the weights are equal, we conclude that the load is completely balanced. Also since

$$cutsize = \sum_{e_j \in \mathcal{N}_\epsilon} = 1 + 1 + 1 = 3,$$

we imply that the communication between processors is minimal. Three parts are shown as the dashed circles in Fig. 8(b).

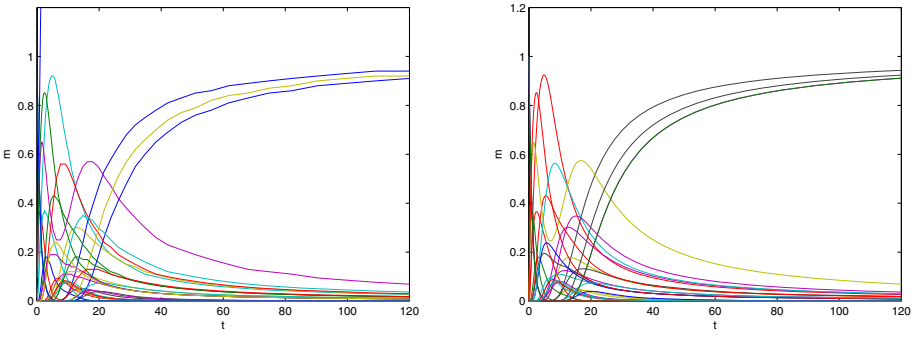
### 4.3 Regrouping Differential Equations

Based on the partition result, the differential equations can be divided into three small equation groups: process $P_2$ and places $m_{32}$, $m_{34}$, $m_{35}$ are in the first equation group; process $P_3$ and the places $m_{13}$, $m_{15}$, $m_{21}$, $m_{22}$ are in the second equation group; process $P_1$ and the places $m_{10}$, $m_{11}$, $m_{12}$, $m_{33}$ are in the third equation group. The follows are the resulting three equation groups:

$$\begin{cases} m_1' = m_9 - m_1 \\ m_2' = m_1 - m_2 \\ m_3' = m_2 - m_3 m_{11} \\ m_4' = m_3 m_{11} - m_4 \\ m_5' = m_4 - m_5 m_{13} \\ m_6' = m_5 m_{13} - m_6 \\ m_7' = m_6 - m_7 m_{15} \\ m_8' = m_7 m_{15} - m_8 m_{33} \\ m_9' = m_8 m_{33} - m_9 \\ m_{10}' = m_2 - m_{10} m_{20} \\ m_{11}' = m_{18} - m_3 m_{11} \\ m_{12}' = m_4 - m_{12} m_{25} \\ m_{33}' = m_{36} - m_8 m_{33} \end{cases}, \begin{cases} m_{23}' = m_{30} - m_{23} \\ m_{24}' = m_{23} - m_{21} m_{24} \\ m_{25}' = m_{21} m_{24} - m_{12} m_{25} \\ m_{26}' = m_{12} m_{25} - m_{14} m_{26} \\ m_{27}' = m_{14} m_{26} - m_{27} \\ m_{28}' = m_{27} - m_{28} m_{35} \\ m_{29}' = m_{28} m_{35} - m_{29} \\ m_{30}' = m_{29} - m_{30} \\ m_{13}' = m_{12} m_{25} - m_5 m_{13} \\ m_{14}' = m_6 - m_{14} m_{26} \\ m_{15}' = m_{29} - m_7 m_{15} \\ m_{21}' = m_{16} - m_{21} m_{24} \\ m_{22}' = m_{21} m_{24} - m_{17} m_{22} \end{cases}, \begin{cases} m_{16}' = m_{10} m_{20} - m_{16} \\ m_{17}' = m_{16} - m_{17} m_{22} \\ m_{18}' = m_{17} m_{22} - m_{18} \\ m_{19}' = m_{39} - m_{19} \\ m_{20}' = m_{19} - m_{10} m_{20} - m_{20} m_{32} \\ m_{31}' = m_{18} + m_{38} - m_{31} \\ m_{36}' = m_{20} m_{32} - m_{36} \\ m_{37}' = m_{36} - m_{34} m_{37} \\ m_{38}' = m_{34} m_{37} - m_{38} \\ m_{39}' = m_{31} - m_{39} \\ m_{32}' = m_{27} - m_{20} m_{32} \\ m_{34}' = m_8 m_{33} - m_{34} m_{37} \\ m_{35}' = m_{38} - m_{28} m_{35} \end{cases}$$

### 4.4 Parallel Computing

Now we employ the PVODE package from SUNDIALS to parallel compute above equations, where the parallel program are distributed in three processors. Meanwhile, we use Matlab to calculate the numerical solution for the original differential equation group. The solutions are displayed in Fig. 9(a) and Fig. 9(b), respectively, where we choose $t = 120$ seconds.




(a) (b)

**Fig. 9.** (a) The solutions of gas station with parallel computing. (b) The solutions of gas station with Matlab.

From the figures, we see that results of two methods are consistent. PVODE solver can not only automatically adjust the step size through the control of local error like the function ODE45 of Matlab, but also reach the maximum step size by periodically adjusting the order. Since here the size of gas station problem is small, the cost of parallel computing is smaller than the cost of communication. Comparing to the time used by serial computing, the time used by parallel computing did not improve much. The advantage of parallel computing will be reflected when the computing size is big enough. The parallel computing algorithm overcomes the drawback of serial computing algorithm regarding computing time, solution accurateness and real time quick response

requirements, etc. Since our parallel program is based on the message passing interface (MPI), its portability, scalability are better than those of data sharing parallel program.

### 4.5 Static Analysis

After computing, we get the following results:

- $m_1(t),\ m_2(t),\ m_3(t),\ m_4(t),\ m_5(t),\ m_6(t),\ m_{10}(t),\ m_{11}(t),\ m_{12}(t),\ m_{13}(t), m_{14}(t), m_{16}(t), m_{17}(t), m_{18}(t), m_{19}(t), m_{20}(t), m_{21}(t), m_{22}(t), m_{23}(t), m_{24}(t), m_{25}(t), m_{26}(t), m_{27}(t), m_{31}(t), m_{32}(t), m_{36}(t), m_{39}(t) \rightarrow 0$,
- $m_7(t), m_{28}(t), m_{33}(t), m_{37}(t) \rightarrow 1$,
- $m_8(t) = m_9(t) = m_{15}(t) = m_{29}(t) = m_{30}(t) = m_{34}(t) = m_{35}(t) = m_{38}(t) = 0$.

We see that all the state measures are fallen to two classes: the state measures either converge to 0 or the state measures converge to 1. From [8], we know that there is deadlock in the program.

## 5 Case Study (II): The Dining Philosopher Problem

Dijkstra's [7] dining philosopher problem is a very well-known example of a concurrent program. Although not a very realistic problem, it does contain a nontrivial deadlock and is probably the most commonly analyzed example. A group of N philosophers is sitting around a table. The philosophers alternate between thinking and eating. Initially n forks are placed on the table between each pair of philosophers. In order to eat, a philosopher must first pick up both forks next to him. The forks are put down when the philosopher finishes eating and starts thinking. This problem is interesting because of the possibility of a circular deadlock. Deadlock may occur if all philosophers pick up their forks in the same order, say, the right fork followed by the left fork. In this case, there is one deadlock state corresponding to the situation in which all philosophers have picked up their right fork and are waiting for their left fork.

In our experiment, we have computed the solutions for several groups of equations when the number of philosophers N = 5, 10, 20, 30 40, 50, 100, 200, 400. Based on our partition process, we have divided each group into 2, 4, 6, and 8 small groups, and then execute them in the computer with 2, 4, 6, and 8 processors. Our experiments are conducted on the computer: 4 CPUs, dual core, AMD Opteron$^{TM}$ Processor 870, CPU 2.0GHZ, memory 16GB.

Table 1 displays the time data for 50 philosophers, 100 philosophers, 200 philosophers, and 400 philosophers that run in 1 processor(serial), 2 processors, 4 processors, 6 processors, and 8 processors, respectively.

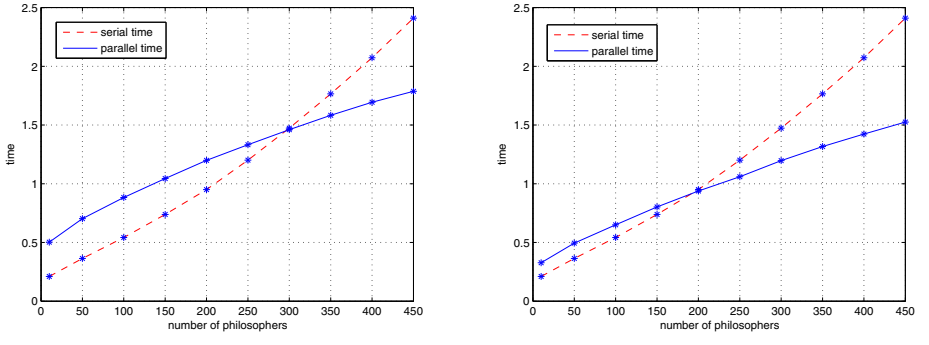
To better see the time increasing trend, we plot the graphs as the following. Fig. 10(a) displays the required time to run in 2 processors when each equation group is divided into 2 groups; Fig. 10(b) displays the required time to run in 4 processors when each equation group is divided into 4 groups.

From the figures, we can analyze the computing time as the the following. 1) The parallel computing time and the serial computing time both will increase as the problem

**Table 1.** Computing time with different number of processors for different number of philosophers

| | serial time(s) | 2-processor(s) | 4-processor(s) | 6-processor(s) | 8-processor(s) |
|---|---|---|---|---|---|
| 50-philosopher | 0.3640 | 0.7032 | 0.4940 | 0.4141 | 0.3859 |
| 100-philosopher | 0.5421 | 0.8829 | 0.6502 | 0.5515 | 0.5156 |
| 200-philosopher | 0.9503 | 1.1996 | 0.9380 | 0.8165 | 0.7667 |
| 400-philosopher | 2.0732 | 1.6942 | 1.4229 | 1.2956 | 1.2389 |

size increases, but the increasing speed for serial is faster than the increasing speed for parallel. If the problem size is small, it is possible that parallel computing time may be bigger than the serial computing time, and thus the computing result may not reflect the advantage of parallel algorithm. For example, in Fig.10(b), when the number of philosophers is less than 200, the parallel computing time is bigger than serial computing time. However, when the number is bigger than 200, the parallel computing time is less than the serial computing time. The point, like 200, at which the parallel computing time starts being less than serial computing time is called Changing Point. 2) As the number of processors increases, the Changing Point is getting smaller, which means that the advantage of parallel algorithm can be reflected in the smaller size problems. 3) If the problem size is fixed, then as the number of processors increases, the parallel computing time is getting smaller.




(a) (b)

**Fig. 10.** (a) Each equation group is divided into 2 groups and are calculated on 2 CPUs. (b) Each equation group is divided into 4 groups and are calculated on 4 CPUs.

Speedup is the second metric to measure the parallel algorithm, which can be calculated by formula: $Sp(n) = Ts(n)/Tp(n)$, where $n$ is the problem size, $Ts(n)$ is the computing time for the fastest serial algorithm in the worst situation, $Tp(n)$ is the computing time for the parallel algorithm in the worst situation. Speedup reflects the degree to which the computing time has been improved when parallel algorithm is used. Table 2 displays the speedup data from our experiment. After analyzing the data, we see that

1) if the problem size is fixed, then as the number of processors increases, the speedup is getting bigger, but the increasing speed will be getting slower; 2) if the number of processors is fixed, then as the problem size increases, the speedup is getting bigger. This indicates that the larger size problem and the more processors being used, the more advantages our parallel algorithm will demonstrate.

**Table 2.** Speedup for different number of processors for different number of philosophers

| | 2-processor | 4-processor | 6-processor | 8-processor |
|---|---|---|---|---|
| 50-philosopher | 0.5176 | 0.7369 | 0.8790 | 0.9432 |
| 100-philosopher | 0.6140 | 0.8338 | 0.9829 | 1.0513 |
| 200-philosopher | 0.7922 | 1.0131 | 1.1639 | 1.2394 |
| 400-philosopher | 1.2237 | 1.4570 | 1.6002 | 1.6734 |

The third matric to measure parallel computing is the parallel efficiency, which can be calculated by formula: $Ep(n) = Sp(n)/P(n)$, where $P(n)$ is the number of processors required to solve the problem, $Sp(n)$ is the speedup. Parallel efficiency reflects the degree to which the processors have been used in the parallel computing. Table 3 displays the obtained efficiency data. After analyzing the data, we see that 1) if the number of processors is fixed, then as the problem size increases, the parallel efficiency will increase. This means that the parallel algorithm is scalable. 2) if the problem size is fixed, then as the number of processors increases, the parallel efficiency is getting smaller. The reason is that as the number of processors increases, the usage rate of each processor and the load to each processor are decreasing, while communication cost is increasing.

**Table 3.** Parallel efficiency for different number of processors and philosophers

| | 2-processor | 4-processor | 6-processor | 8-processor |
|---|---|---|---|---|
| 50-philosopher | 25.88% | 18.42% | 14.65% | 11.79% |
| 100-philosopher | 30.70% | 20.85% | 16.38% | 13.14% |
| 200-philosopher | 39.61% | 25.33% | 19.40% | 15.49% |
| 400-philosopher | 61.19% | 36.43% | 26.67% | 20.92% |

## 6 Related Work

The first application of hypergraph partitioning in the domain of performance analysis is by Dingle et al. [9]. Their method utilized hypergraph partitioning presented in [5] to minimize inter-processor communication while maintaining a good load balance. They demonstrated the approach by calculating passage time densities in a 1.6 million state Markov chain derived from a Generalized Stochastic Petri net model and a 10.8 million state Markov chain derived from a closed tree-like queueing network. However, the computing is still limited by the compute power and RAM provided by a network of workstations.

Alimonti and Feuerstein [1] presented a work to translate Petri net to Hypergraph. This translation keeps more information of Petri net. It has been used to prove that the liveness and boundedness of Conflict-free net can be solved in a polynomial time. However, since the translation does not heavily decrease the size of the Petri net, the resulting hypergraph is not proper for parallel computing.

The parallel solution of initial value problems for ordinary differential equations has received much interest from many researchers in the past decades. Efforts have been taken to find efficient parallel solution methods, e.g., extrapolation methods [10], waveform relaxation techniques [3], and iterated Runge-Kutta methods [15]. Most of these approaches develop new numerical algorithms with a larger potential for parallelism, but with different numerical properties. Rao and Mouney [18] have considered the problem of data communication in parallel block predictor-corrector (P-BPC) methods for solving ODE's using only time as well as time and space discretizations for systems of equations. In our case, for stiff problems, PVODE employs Newton-Krylov linear iteration. This technique exhibits good convergence behavior and has become increasedly competitive with classical iterative methods in terms of memory utilization. Finally, the method is well suited to implementation on parallel computers.

## 7 Conclusion

This paper presented a method to parallel compute a class of ODEs that describe concurrent systems. In our numerical experiment, we have used two examples to check the feasibility of our method for the performance analysis of concurrent systems. In the future, we will apply our method to different sizes of concurrent systems to experience the the advantages of parallel computing that decrease the memory and the computing labor, and determine when to use serial computing, parallel computing or their combination. Further more, we may consider the combination of parallel computing for time and space if the size of the problem is large enough.

## References

1. Alimonti, P., Feuerstein, E.: Petri Nets, Hypergraphs and Conflicts. In: van Leeuwen, J. (ed.) WG 1993. LNCS, vol. 790, pp. 293–309. Springer, Heidelberg (1994)
2. Burch, J.R., Clarke, E.M., Long, D.E.: Representing Circuts More Efficiently in Symbolic Model Checking. In: Proceedings of the $28^{th}$ Design Automation Conference, pp. 403–407. IEEE Computer Society Press, Los Alamltos (1991)
3. Burrage, K.: Parallel and Sequential Methods for Ordinary Differential Equations. Oxford Science Publications (1995)
4. Çatalyürek, Ü., Aykanat, C.: A Hypergraph-partitioning Approach for Coarse-grain Decomposition. In: ACM/IEEE Supercomputing, Denver (2001)
5. Çatalyürek, Ü., Aykanat, C.: Hypergraph-partitioningbased Decomposition for Parallel Sparse-matrix Vector Multiplication. IEEE Transactions on Parallel and Distributed Systems 10(7), 673–693 (1999)
6. David, R., Alla, H.: Autonomous and Timed Continuous Petri Nets. In: Proceedings of $11^{th}$ Intl Conference on Application and Theory of Petri Nets, Paris, France, pp. 367–381 (1990)
7. Dijkstra, E.W.: Hierarchical Ordering of Sequential Processes. Acta Informatica 2, 115–138 (1971)

8. Ding, Z.: Static analysis of concurrent programs using ordinary differential equations. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 1–35. Springer, Heidelberg (2009)
9. Dingle, N.J., Harrison, P.G., Knottenbelt, W.J.: Uniformization and Hypergraph Partitioning for The Distributed Computation of Response Time Densities in Very Large Markov Models. Journal of Parallel and Distributed Computing 64, 908–920 (2004)
10. Ehrig, R., Nowak, U., Deuflhard, P.: Massively Parallel Linearly-implicit Extrapolation Algorithms As a Powerful Tool in Process Simulation. Parallel Computing: Fundamentals, Applications and New Directions, 517–524 (1998)
11. Hiraishi, K.: Performance evaluation of workflows using continuous petri nets with interval firing speeds. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 231–250. Springer, Heidelberg (2008)
12. Helmbold, D., Luckham, D.: Debugging Ada Tasking Programs. IEEE Software 2(2), 47–57 (1985)
13. Hendrickson, B., Kolda, T.G.: Graph Partitioning Models for Parallel Computing. Parallel Computing 26, 1519–1534 (2000)
14. Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S.: SUNDIALS, Suite of Nonlinear and Differential/Algebraic Equation Solvers. ACM Transactions on Math. Softw. 31, 363–396 (2005)
15. van der Houwen, P.J., Sommeijer, B.P.: Parallel Iteration of High-order Runge-Kutta Methods With Stepsize Control. J. Comput. Appl. Math. 29, 111–127 (1990)
16. Karypis, G., Aggarwal, R., Kumar, V., Shekhar, S.: Multilevel Hypergraph Partitioning: Application in VLSI Domain. IEEE Transactions on VLSI Systems 7(1), 69–79 (1999)
17. Lengauer, T.: Combinatorial Algorithms for Integrated Circuit Layout. John Wiley & Sons, Inc., New York (1990)
18. Rao, P.S., Mouney, G.: Data Communication in Parallel Block Predictor-corrector Methods for Solving ODE's. Parallel Computing 23, 1877–1888 (1997)
19. Silva, M., Recalde, L.: Continuization of timed petri nets: From performance evaluation to observation and control. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 26–47. Springer, Heidelberg (2005)

# High-Performance Reconfigurable Computer Systems

Alexey Dordopulo, Igor Kalyaev, Ilya Levin, and Liubov Slasten

Kalyaev Scientific Research Institute of Multiprocessor Computer Systems
at Southern Federal University,
Chekhov street 2, 347922 Taganrog, Russia
{scorpio,kaliaev,levin}@mvs.sfedu.ru,
lmslasten@yandex.ru

**Abstract.** The FPGA-based reconfigurable computer systems have high real performance and provide practically linear growth of performance when hardware system resource is growing. The paper deals with design features, technical characteristics and values of real performance of computational modules of reconfigurable computer systems, designed on the base of Virtex-6 FPGAs. In addition, a software suit, intended for development of parallel applications for the RCS, is considered.

**Keywords:** reconfigurable systems architecture, multiprocessor computer systems, supercomputers, parallel processing, pipeline processing, high performance, computer-aided circuit design, structure-oriented languages, communication system, hardware.

## 1 Introduction

Until recently supercomputer technology had been developed according to large-block integration principles. Performance was provided owing to technological achievements in the area of microprocessor design and communication system development, but advantages of architecture and circuit design were not used. As a result, cluster systems, designed on the base of widely-available computer nodes and communication networks, became widespread during the last years. Cluster systems have relatively low price and simple programming methods. When solving loosely-coupled problems, clusters provide high real performance. However, real performance considerably reduces, if the problem is tightly-coupled. In addition, cluster system performance reduces when number of processors in cluster supercomputer grows. This is the result of dissimilarity between information structure of the problem and cluster "hard" architecture. Real performance of supercomputer decreases and burden sharply increases, if the solving problem is tightly-coupled and requires a great number of data exchange operations. The concept of multiprocessor computer systems with reconfigurable architecture had proved successful in solving specified problems of cluster supercomputers [1, 2, 3].

The essence of this concept is adaptation of reconfigurable computer system (RCS) architecture to the structure of information graph of the solving problem. Using principles of the concept, we can considerably reduce computational burden and increase system real performance.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 272–283, 2011.

The concept of the RCS design has been developed during decades. In 80s-90s of the 20th century several prototypes of computer systems with reconfigurable architecture were created. Nevertheless, lack of suitable element base restrained wide use of the RCSs during a long time. The RCS element base is to fulfill the following requirements: ability of hardware (structural) implementation of large computational fragments; programming (reconfiguration) of various computational structures which fit to the active problem; ability of usage of computer-aided design (CAD) systems, software development environments and software tools for computational structure reconfiguration; acceptable price, etc.

Field Programmable Gates Array (FPGA) with grand-scale integration fulfills all these requirements [4-6]. Capabilities of internal structure reconfiguration are the distinctive feature of the FPGAs. Owing to this, the FPGAs are the best choice for the concept of reconfigurable computer system design.

At present the FPGAs are ever more widely used as components of supercomputers for their real performance increasing. Let us give two typical examples. In 2007 Silicon Graphics launched a supercomputer which contained 35 modules RC100 with two FPGAs in each module. According to information, represented by Silicon Graphics, the supercomputer solves bioinformatics problems in 900 times faster, than a cluster which contains 68 nodes, designed on the base of Opteron processors. A new product of Silicon Graphics is module SGI RC200-blade which contains the FPGAs Altera Stratix III and Intel QuickAssist [7] technology of direct interface between FSB and Xeon processors.

In 2009 Cray launched a supercomputer Cray XT5h [8], which combines scalar and vector processors and the FPGA processors. Scalar processors are perfectly suited for calculations with few data access operations; the FPGAs and vector processors are usually used for solving problems which require high memory loading and large volumes of processed data. Modern complicated computational problems may be solved much more effectively by using a supercomputer which contains all these types of processors. The peak performance of the supercomputer Cray XT5h is 2331.00 Tflops (the 1st position in the TOP500 list).

However, almost in all supercomputers the FPGAs are used as co-processors to standard computational nodes, implemented on general-purpose microprocessors. Owing to the concept of the RCS design, it is possible to use the FPGAs as an element base for creation of large computational fields, which may be reconfigured into a set of small-grained special-purpose computing structures, adapted to the structure of the problem. Various computing structures, similar to the information structure of the solving problem, may be created and fine-tuned within large computational field. The approach ensures maximization of system real performance during execution of the problem.

The main principles of the RCS design, based on the FPGA fields, are the following:

1. The FPGAs are combined into a computational field.
2. A multipipeline computational structure, similar to the informational graph of the solving problem, is designed within the FPGA field using CAD and structural programming software tools.

3. If the informational graph of the solving problem can not be mapped on the FPGA field because of hardware constraints, it must be partitioned into disjoint subgraphs. Each subgraph must be mapped on the FPGA field, i.e. structurally realized. Sequential procedure of subgraph structural realization within the FPGA field is organized after mapping (Fig.1).

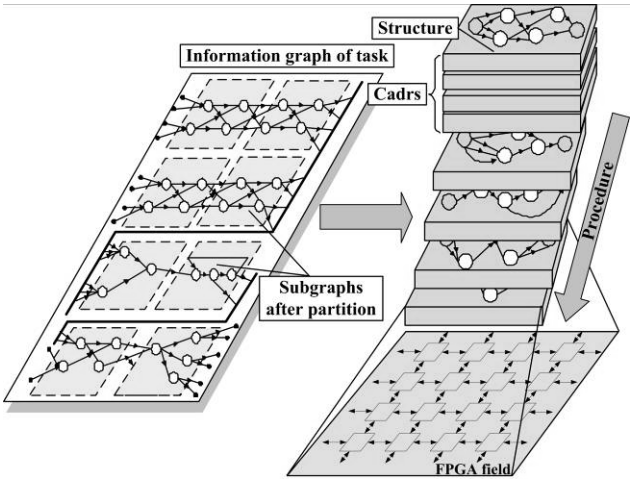



**Fig. 1.** Structural-procedural organization of calculations in the FPGA field

4. It is evident that the larger is the FPGA field, the smaller number of subgraphs is obtained after partition of the informational graph. As a result, some burden in the FPGA field reconfiguration may be reduced and the RCS real performance at solving the problem may be increased.

The RCS, designed for solving complicated problems, has to contain hundreds and thousands of the FPGAs of large-scale integration, combined in the computational field. It is clear, that placing of such number of the FPGAs on a single circuit board is impossible. This problem can be solved by using a principle of modular design of the RCS computational field on the base of unified basic modules. Basic module is a board which contains a fragment of the FPGA computational field and auxiliary elements such as intermodule data exchange interfaces, distributed memory blocks, secondary power units, synchronization subsystem, control nodes, network interfaces, etc. Basic module is a small-sized RCS, which is able, along with a personal computer, to solve user problems. Several basic modules may be united in a single RCS with the needed performance. Generalized structure of the RCS basic module is given in Fig. 2.

The main computational abilities of the basic module are concentrated in its computational field, which contains a set of large-scale integration FPGAs. Distributed memory blocks are implemented on the base of standard chips SRAM or SDRAM of the needed capacity and transfer rate. Control of the basic module resources is performed by basic module controller (BMC). Such procedures as initial data loading and result data unloading, loading fragments of parallel applied programs into distributed memory controllers are also realized by the BMC.

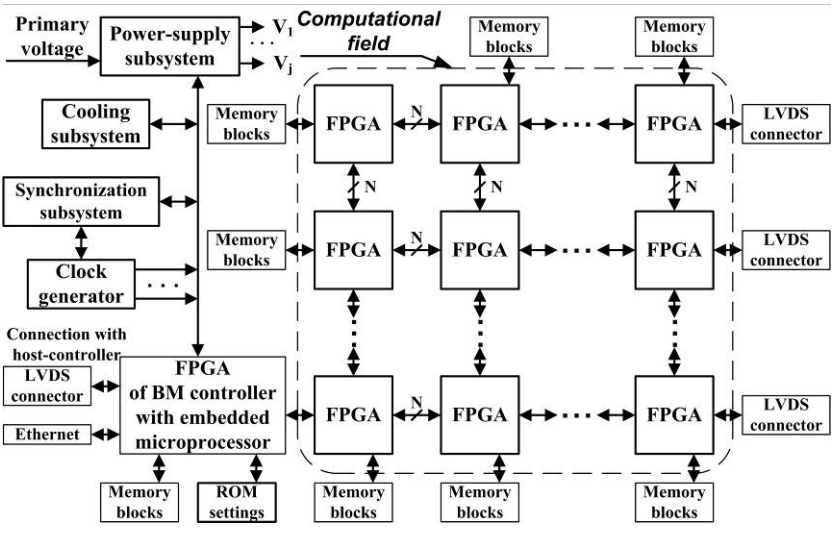



**Fig. 2.** Structure of the RCS basic module

The FPGA connections within the computational field are to provide high data transfer rate between parts of computational structures placed in other microchips of the computational structure multichip implementation. Therefore connections between the FPGAs have a minimum length and are implemented using the LVDS or RocketIO technologies.

Advantages of high speed interfaces are the following: low power consumption of output cascades, low level of electromagnetic radiation, non-susceptibility to common-mode electromagnetic radiation interference and hardware support of high rate data transfer in modern FPGAs. Each line of the high speed interface is a pair of differential strip conductors connected to special pins of microchips. The data transfer rate for each two-wire transmission line depends on the implementation and is up to 5 Gbit/sec. A detailed analysis of the general principles of RCS organization and functioning is given in [1, 2, 3].

## 2 Examples of the RCS Implementation

Some types of the RCS basic modules (BM), designed in Kalyaev Scientific Research Institute of Multiprocessor Computer Systems of Southern Federal University and Scientific Research Centre of Supercomputers and Neurocomputers (SRI MCS SFU, Taganrog, Russia), are given in Fig.3.

A family of the RCSs of various performance – from 50 Gflops (16 Virtex-5 FGPAs) to 6 Gflops (1280 Virtex-5 FPGAs) – was designed on the base of BM 16V5-75 [3, 9].

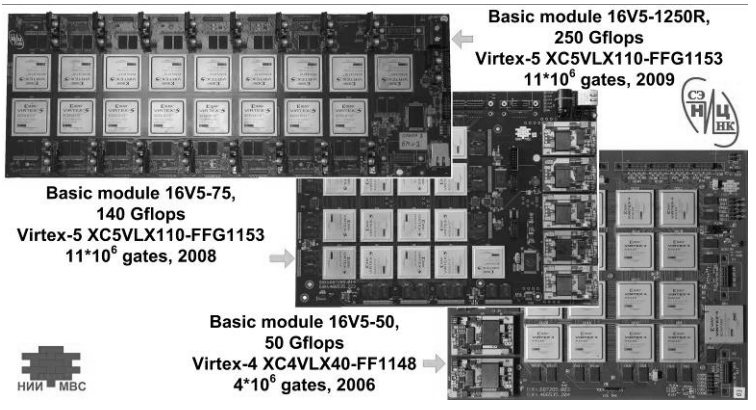



**Fig. 3.** The RCS basic modules

RCS-5, the representative of the RCS family, contains five 19' ST-1R racks. RCS-1R contains one 19' ST-1R rack. Each rack contains four blocks RCS-0.2-CB designed according to "Euromechanics" 6U standard. Each block may contain up to four basic modules 16V5-75 with peak performance up to 200 Gflops each for single precision data. Connections within the computational field 16V5-75 are implemented according to the LVDS standard and provide total data transfer rate over 3 Tbit/sec at frequency of 1200 MHz.

In 2010 specialists of SRI MCS SFU designed modular scalable RCS with performance of 20 Tflops. The RCS is placed in 19' rack and contains 1536 FPGAs Virtex-5. The RCS is designed on the base of the BM 16V5-1250R with performance of 250 Gflops. High speed LVDS channels connect all basic modules into a unified computational resource.

Four basic modules 16V5-1250R are placed in a block Orion-5, designed according to "Euromechanics" 1U standard. All FPGA chips of the computational field of the basic module 16V5-1250R have LVDS channels for computational field expansion in contrast to the basic module 16V5-75, which has only four of sixteen chips with expansion ability. This distinction provides high data exchange rate between basic modules of the block Orion-5.

Real performance of RCS-0.2-CB and Orion-5 at solving problems of various classes, such as digital signals processing, linear algebra and mathematical physics, are given in the table 1.

**Table 1.** Performance of RCS-0.2-CB and Orion -5

| Tasks | DSP, Gflops | Linear algebra, Gflops | Mathematical physics, Gflops |
|---|---|---|---|
| RCS-0.2-CB | 647.7 | 423.2 | 535.2 |
| Orion-5 | 809.6 | 528.7 | 669.0 |

Values of the table 1 show that the computational blocks Orion-5 and RCS-0.2-CB have almost equal computational field resources, but Orion-5 has higher performance. This is explained by high capacity of the channels, which connect computational fields of basic modules, and by larger number of distributed memory channels connected to the computational field: 80 channels in RCS-0.2-CB and 128 channels in Orion-5.

Values of performance per unit volume of these two blocks are rather different. For Orion-5 this value is higher because it has more rational design. Volume of Orion-5 is 18% from total volume of RCS-0.2-CB or 32% from its computational part. Table 2 contains values of specific performance of RCS-0.2-CB and Orion-5 at solving problems using single precision floating-point data. High specific performance of Orion-5 is very important for design of multirack high performance RCSs.

**Table 2.** Specific performance of the blocks

| Performance | General problems, Gflops/dm$^3$ |
|---|---|
| RCS-0.2-CB | 16.7 |
| Orion-5 | 64.9 |

Orion-5 and the rack ST-1R have no external high rate LVDS-channels for resource expansion of the computational fields. Performance of the systems which consist of the blocks Orion-5 and the racks ST-1R may be increased by using network technologies with the help of Ethernet channels, as it is made in RCS-5 [2, 3, 9].

At the beginning of 2011 scientists of SRI MCS SFU have produced the RCSs of new generation, which are created according to the principles of open scalable architecture and consist of computational modules, designed on the base of Virtex-6 FPGAs. The circuit boards are basic components of two perspective computational modules Saiph and Rigel, named after the stars of constellation of the Orion. Table 3 contains specifications of the circuit boards of Saiph (6U standard) and Rigel (1U standard), intended for placing into a standard 19" computational rack, which is a base component of super high-performance FPGA-based complexes.

**Table 3.** Specifications of the circuit boards of the computational modules

| Circuit board of computational module | Number of the FPGAs | FPGA type | Number of equivalent gates in one FPGA, million | Interface and rate of data exchange between the modules, Gbit/sec | Power consumption, Watt |
|---|---|---|---|---|---|
| Orion-5 | 16 | Virtex 5 | 11 | LVDS, 1.2 | 250 |
| Saiph | 8 | Virtex 6 | 24 | Gigabit Ethernet, 1 | 300 |
| Rigel | 8 | Virtex 6 | 24 | Gigabit Ethernet, 1 | 300 |

Owing to the Virtex-6 FPGAs, used as an element base of Saiph and Rigel, performance of the computational modules may be increased in 1.5-2 times in comparison with Orion-5, which was designed on the base of the Virtex-5 FPGAs. At the same time price of the computational modules remains unchangeable. Hence, the

designed computational modules of the new generation may be considered as the most perspective for design of the RCSs of various architectures and configurations. Engineering characteristics, such as specific performance, power effectiveness, etc., are considerable advantages of the modules.

Table 4 contains specifications and values of peak performance of considered computational modules and racks, which use the modules as computational components. Performance of modules and racks was estimated during processing of data with single and double precision, represented in IEEE-754 format.

**Table 4.** Performance of the computational modules and racks

| Name of computational module | Performance of computational module ($Pi_{32}$/$Pi_{64}$), Gflops | Number of computational modules in a 19" rack | Performance of rack ($Pi_{32}$/$Pi_{64}$), Tflops |
|---|---|---|---|
| Orion-5 | 1000/340 | 24 | 24/8.1 |
| Saiph | 1600/500 | 6 | 9/3 |
| Rigel | 1600/500 | 24-36 | 34.5 – 51.8 |

Table 5 contains values of computational modules performance, obtained during solving problems of mathematical physics with single precision floating-point data.

**Table 5.** Performance of the computational modules

| Name of computational module | Mathematical physics, floating point arithmetic, Tflops |
|---|---|
| Orion-5 | 1/0.34 |
| Saiph | 1.6/0.5 |
| Rigel | 1.6/0.5 |

Table 6 contains values of total data transfer rate between the FPGAs and distributed memory units, and between the FPGAs of different computational modules.

**Table 6.** Data transfer rate

| Name of computational module | With distributed memory units (Gbit/sec) | Between the FPGAs of computational field (Tbit/sec) | With other computational modules (Tbit/sec) |
|---|---|---|---|
| Orion-5 | 12.8 | 1.2 | 1.2 |
| Saiph | 12.8 | 1.0 | 1.0 |
| Rigel | 12.8 | 1.0 | 1.0 |

In comparison with the RCS Orion-5, computational modules of new generation Saiph and Rigel, designed on the base of Virtex-6 FPGAs, may be used as basic components of new computer systems. At the same time the computational modules may be used as standalone systems or as accelerators of the IBM PC.

## 3 Reconfigurable Computer Systems Programming

Process of software development for the RCS differs a lot from that for multiprocessor computer systems with cluster architecture. Software development for the RCS may be divided into two parts: structural programming and procedural programming. The results of structural programming are computational structures within the field of the FPGA logical cells, which are required for calculations. The sense of procedural programming is very similar to traditional programming and consists in creation of computational process in the RCS. Structural programming of the FPGA computational field is the most difficult for the RCS programmer, because traditional programming consists only in creation of computational process based on fixed hardware. For programming of computational structures, designed on the base of the FPGAs, the programmer needs quite different skills – skills of a circuit engineer.

A special software complex for applied program development helps to make the RCS programming more simple. Using this complex, software programmer can develop applications without any special knowledge in area of the FPGA circuit design [1, 3, 10]. The software complex allows:

- to program structural and procedural components using high level programming language COLAMO;
- to develop and to modify computational structures for applications without high qualified circuit engineer;
- to provide applications portability for the RCSs with various architectures;
- scaling of applications at resource extension;
- remote usage and monitoring of the RCS computational resources.

According to functions the software complex may be divided into two parts:

- tools for application development;
- tools for the RCS computational resource control and administration.

The tools for application development contain: integrated development environment Argus IDE with support of Argus and COLAMO programming languages; translator of the RCS high level programming language COLAMO; translator of Argus assembler language; development environment of circuit solutions Fire!Constructor for synthesis of scalable parallel-pipeline structures, which is based on IP-cores library (library of computational structures and interfaces).

A general structure of the RCS application development is given in Fig. 4.

Both structural and procedural components of the application are developed with the help of high level programming language COLAMO, which also allows to create and to modify computational structures of the application without involving high qualified circuit engineer. Applications are portable to the RCSs of various architectures owing to libraries of descriptions of basic modules, blocks, the RCSs (the RCS passport).

The language contains no explicit forms of parallelism description. The solving problem is parallelized by means of access to variables and by indexing of array elements. Conflicts of concurrent reading and writing from/into memory cells within the current cadre may be eliminated by means of widely-used rule the only substitution, according to which a variable stored in memory may be assigned a value only once within the cadre.

Owing to implicit description of parallelism, defined by access type, granularity of program parallelism may be easily controlled on level of data structures description. In addition, application programmer may easily create concise descriptions of various types of parallelism.

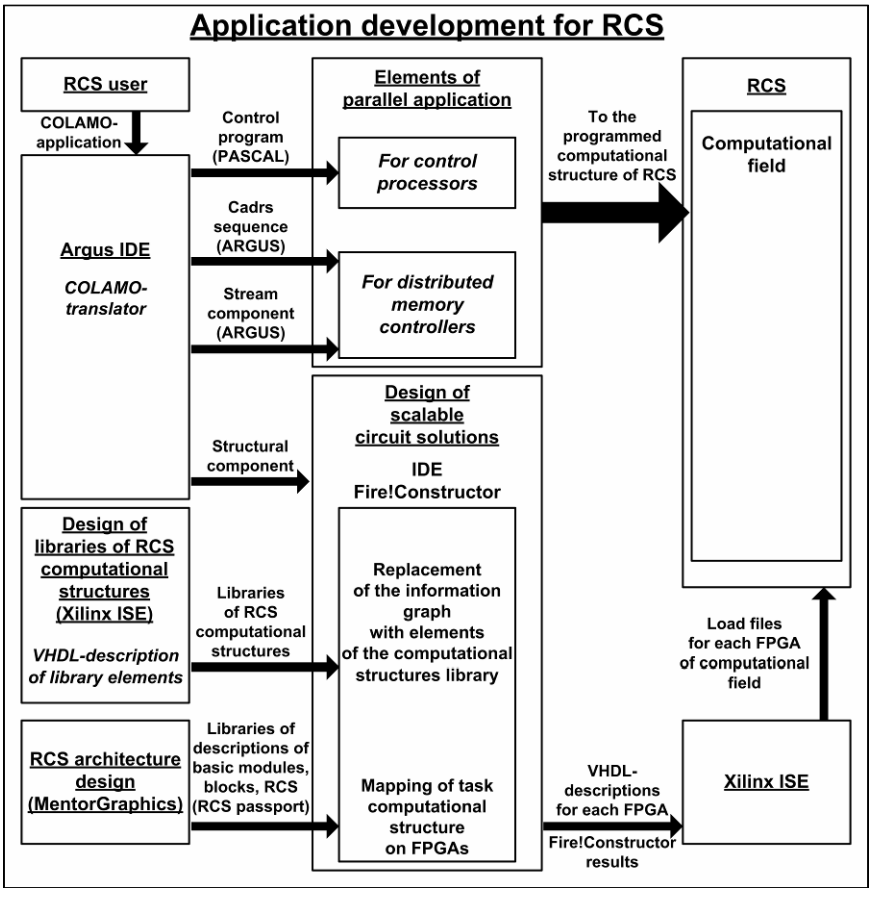



**Fig. 4.** General structure of the RCS application development

There are two base methods of data access: parallel access defined by type "Vector" and sequential access defined by type "Stream". Figure 5 shows programs with different declarations of types of parallelism and graphs of the synthesized computing structures.

Access type "Stream" points out sequential processing of elements of one-dimensional array. Owing to access type "Vector" one-dimensional array elements may be processed concurrently.

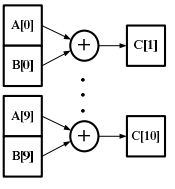
Each dimension of multidimensional arrays may have sequential or parallel access, defined by key word Stream or Vector, respectively.

The result of COLAMO translator is four components of parallel application: control, procedural, stream and structural.

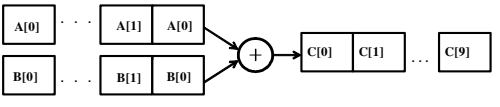
Control component is translated into Pascal language and is executed in master controllers, which, depending on the RCS hardware platform, are contained in basic modules or/and computational blocks. Control component provides loading of configuration of computational FPGAs, initial data loading and results unloading from distributed memory of computational fields. It also provides data exchange between computational blocks or basics modules.

```
VAR A,B,C: Integer [10 : Vector]
Mem;
VAR I : Number;
    CADR SummaVector;
        For I := 0 to 9 do
            C[I] :=A[I]+B[I];
    ENDCADR;
```

```
VAR A,B,C : Integer [10 : Stream] Mem;
VAR I : Number;
        CADR SummaStream;
            For I := 0 to 9 do
                C[I] :=A[I]+B[I];
        ENDCADR;
```




a)




b)

**Fig. 5.** Parallel and sequential addition of arrays

Procedural and stream components are translated into Argus assembler language. Like control component, both of these components are elements of parallel application. Procedural and stream components are executed by distributed memory controllers, which specify the sequence of cadr execution and create parallel data flows in cadr computational structures.

One of the most interesting features of COLAMO translator is a function of extraction of structural component from COLAMO-application. The translator generates structural component in object form, which describes an informational graph of calculations in a cadr. Using this object form, the development environment of scalable circuit solutions Fire!Constructor synthesizes a computational structure for all FPGAs of the computational field.

The environment Fire!Constructor is completely new software product, which has no analogues in the world. Initial data for Fire!Constructor are the following:

– structural component of COLAMO-application in object form, generated by the COLAMO-translator;

– library of computational structures and interfaces (the IP-cores), used for informational graph mapping on the RCS hardware. The IP-cores of computational structures and interfaces are designed by circuit engineers during the RCS design and creation. Then the VHDL-descriptions of the IP-cores are included into the library which is used by Fire!Constructor;

– library of descriptions of basic modules, blocks and the RCS (the RCS passport).

Using the RSC passport library, Fire!Constructor generates platform-independent multichip structures within the FPGA computational fields of basic modules, blocks and the RCS racks, including heterogeneous RCSs, which consist of various basic modules and blocks.

Fire!Constructor automatically synchronizes data flows within a multichip implementation of parallel-pipeline cadr computational structure [11-14] according to features of the RCS computational fields: number and types of connections between the FPGAs of basic modules and connections between computational fields of basic modules, blocks and racks. Fire!Constructor generates the following result information for each FPGA of the RCS computational field:

– the VHDL-description (*.vhd file) of fragment of cadr computational structure within the FPGA;

– information of correspondence (*.ucf) between logical names of external signals of computational structure fragment and pins of the FPGA.

Using result information, obtained from Fire!Constructor, Xilinx ISE generates load configuration files (*.bit) for all FPGAs of the computational field.

For the RCS programming all configuration files must be loaded into the FPGAs of the computational field and all components of the parallel application must be loaded into master controllers and distributed memory controllers.

The software complex allows application programmer to develop effective applications for the RCS for solving problems from various problem areas and provides easy programming and automatic transfer of structural solution from one RCS architecture into another. Usage of software complex reduces cost (in 2-3 times) and time (in 3-5 times) of application development in comparison with traditional method, when the RCS computational structure is designed by a circuit engineer.

## 4 Conclusions

Reconfigurable computer systems, designed on the base of the FPGAs, are a new direction of high-performance technique development. In contrast to cluster supercomputers the RCSs allow to create, within basic architecture, virtual special-purpose calculators with structure similar to the structure of the solving problem. This provides high effectiveness of calculations and almost linear performance growth at computational resource expansion.

General-purpose architecture of the RCSs, based on the FPGA computational fields, which can be reprogrammed according to the structure of the solving problem, gives wide possibilities of usage of the RCSs in the problem areas, which require:

- highly effective computer facilities, like special-purpose computer systems;
- ability of solving problems from various problem areas.

The RCSs with general-purpose basic architecture and ability of architecture reconfiguring entirely satisfy these requirements.

## References


1. Kalyaev, A.V., Levin, I.I.: Modular scalable multiprocessor systems with structural procedural organization of calculations. Yanus-K, Moscow (2003) (in Russian)
2. Kalyaev, I.A., Levin, I.I., Semernikov, E.A., Shmoilov, V.I.: Reconfigurable multipipeline computational structures. SSC-RAS publishing, Rostov-on-Don (2009) (in Russian)
3. Dmitrenko, N.N., Kalyaev, I.A., Levin, I.I., Semernikov, E.A.: Family of multiprocessor computer systems with dynamically reconfigurable architecture. Bulletin of computer and information technologies 1(6), 2( 7), 2–8, 2–10 (2009) (in Russian)
4. http://www.dinigroup.com
5. http://www.copacobana.org
6. http://www.sciengines.com
7. iXBT.com, http://www.ixbt.com/news/hard/index.shtml?09/60/92
8. Informational analytical centre, http://www.parallel.ru/
9. Kalyaev, I.A., Levin, I.I., Semernikov, E.A.: Family of computer systems with high real performance on the base of FPGA. In: Parallel computer technologies (PaCT 2010): Papers of International Scientific Conference, Ufa, March 29 - April 2, pp. 199–210 (2010) (in Russian)
10. Levin, I.I.: High level language of parallel programming for structural procedural organization of calculations. In: Papers of All-Russian scientific conference, pp. 108–112 (2000) (in Russian)
11. Hendrickson, B., Kolda, T.: Graph partitioning models for parallel computing. Parallel Computing Journal (26) (2000)
12. Hendrickson, B., Leland, R.: Multidimensional spectral load balancing. Sandia National Laboratories, Albuquerque (1993)
13. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal of Scientific Computing 20(1), 359–392 (1998)
14. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: Proceedings of the 1995 ACM/IEEE Supercomputing Conference, pp. 658–677. ACM/IEEE (1995)

# CacheVisor: A Toolset for Visualizing Shared Caches in Multicore and Multithreaded Processors

Dmitry Evtyushkin[1], Peter Panfilov[1], and Dmitry Ponomarev[2]

[1] Moscow State Institute of Electronics and Mathematics (Technical University), Department of Computer Systems and Networks, B. Trekhsvyatitelski per. 3, Moscow 109028
evt.dim@gmail.com, panfilov@miem.edu.ru
[2] State University of New York at Binghamton, Department of Computer Science, Binghamton, NY 13902-6000
dima@cs.binghamton.edu

**Abstract.** In this paper, we present CacheVisor – a toolset for visualizing the behavior of shared caches in multicore and multithreaded processors. CacheVisor uses the memory access traces generated by the execution-driven processor simulation to graphically depict the cache sharing dynamics among applications that concurrently use the cache. We present the implementation of CacheVisor and describe how it can be used in computer architecture research and education. The public release of CacheVisor is planned in the near future.

## 1 Introduction

Microprocessor design industry has recently undergone a fundamental transition from single-core to multicore architectures [1,2,3,4]. While current implementations of multicore chips include a modest number of cores, various sources such as Intel and Berkeley predict hundred [2] or even thousand [1] of cores per chip in the future. In fact, doubling of the number of cores per chip every 18 months is dubbed by some researchers as the new Moore's Law [1]. In addition, each core often support Simultaneous Multithreading (SMT), allowing multiple programs to execute concurrently. The multicore revolution has shifted the focus of the research community from the individual core designs (the cores are rapidly becoming commodities) to the design of shared cache hierarchies and the memory subsystem [1,2,3,4].

The optimal utilization of the shared cache hierarchies in SMT and multicore processors is critical to the performance of these emerging systems. Most of the existing research on this topic is based on empirical analysis of cycle-accurate simulation results, generated mainly in the form of various "average-case" statistics [8,9]. Typically, a benchmarking program of interest (or its representative sample of several hundred million instructions) is simulated, and average performance estimates are computed. For caches, these can be in form of cache hit rates, MPKI (Misses per Kilo Instruction), cache utilization and so on. These statistics are then used to guide the design decisions and implement possible performance optimizations. Unfortunately, average values provide little insight into the dynamic behavior of the

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 284–289, 2011.

cache memory system, and therefore they are of limited use for many important studies, such as investigating dynamic adaptation of caches, or dynamic sharing of cache resources among competing applications (which is the main focus of this paper). As a result, many of the proposed cache optimizations are "ad-hoc" in nature in that they are based on the limited observations of the cache statistics. Better cache sharing algorithms could be developed if the designers had an opportunity to observe the cache sharing patterns in real time, as the program executes.

To address this issue and help the processor designers better understand the behavior of shared caches in multicore and multithreaded systems, we propose to augment the existing state-of-the-art cycle-accurate processor simulators with the visualization backend to enable visual introspection of the memory system activities in real time. In this paper, we demonstrate the application of this approach to the simulation of the cache hierarchy in multithreaded and multicore processors. Specifically, we show how the new visualization environment (called CacheVisor in the rest of the paper) can track the dynamic cache sharing between multiple applications co-executing on a multicore or multithreaded processor. We developed CacheVisor on top of M-Sim simulator [5]. M-Sim (publicly available at http://www.cs.binghamton.edu/~msim) is an extension of a widely-used Simplescalar simulator that supports SMT and multicore simulations.

## 2 CacheVisor Implementation

In general, two approaches can be used for interfacing the output of the cycle-accurate processor and cache simulator with the visualization back-end.

**Offline Approach.** In this approach, the relevant simulation results and statistics are first generated and saved into a trace file. Then, the trace file is read by the visualizer. Here, the visualizer is essentially decoupled from the main simulation engine and can be supplied as an independent module. All that is needed is a clear interface to the trace file. The limitation of this approach is that the amount of information to be visualized is limited by the size of the trace file.

**Online Approach.** In this case, the results of the simulation are generated every cycle (or every few cycles) and are immediately supplied to the visualizer that runs in parallel with the simulator. The visualizer then immediately depicts any changes of the resource usage and in other relevant statistics. This approach couples the visualizer and the simulator more tightly and they are supplied in a unified package.

Both approaches have their advantages and limitations. While the online approach avoids complications associated with storing and retrieving simulation traces, the offline approach requires minimal synchronization between the components and is therefore easier to implement. In addition, decoupling of the visualizer code from the main simulator code also makes the visualizer more portable to other architectures and simulators. The main problem with offline approach is a large size of the trace file, as well as the performance of the visualizer, since it needs to handle considerable amounts of trace information.

The current version of CacheVisor was designed using the offline approach. In this design, the entire visualization system is separated into two parts: cache statistics

collection module and standalone visualizer. The cache statistics collection module is a piece of code embedded in M-Sim simulator, which provides basic facilities for collecting information for every cache event happening during the simulation and storing the information about this event into a trace file. While the implementation of the cache statistics collection module depends significantly on a particular simulator, CacheVisor itself is not tied up to a specific simulator; it uses a fairly simple and portable format for storing the data associated with cache access events.

The general architecture of CacheVisor implemented using offline approach is shown in Figure 1. As the cache accesses occur during the simulation, the cache statistics collection module tracks these accesses. After the cache event occurs, the statistics collection module checks the type of the event (hit or miss), the id of the application that is making the request, the current state of the accessed cache line, and makes the decision of whether or not to write the information about this particular access into a trace file. After the completion of the simulation, the visualizer simply works with the created trace file without using the simulator. Some significant filtering of the events can be done in this step. For example, to visualize the cache capacity sharing between two applications, it is sufficient to only capture the accesses where a cache miss by one thread evicts the cache line currently owned by another thread. This is the only scenario that redistributes the cache space between the two threads. In all other cases, such as when a thread hits in the cache or it evicts another cache line that it already owns, the number of cache lines allocated to each thread does not change. Therefore, those events are insignificant for the visualization.

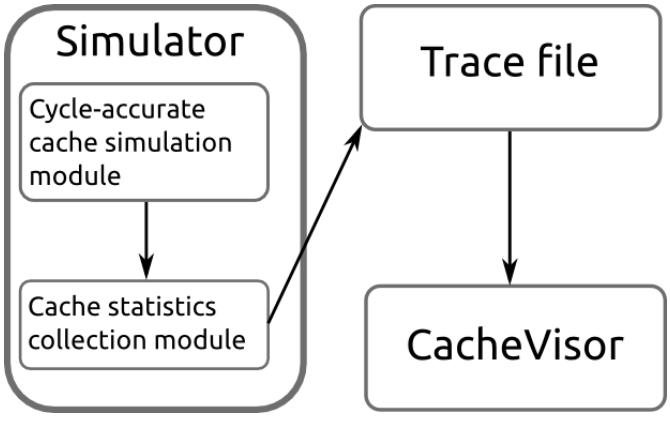



**Fig. 1.** Interaction of CacheVisor and M-Sim using the trace file

The trace file read by CacheVisor is generated in text format, making the trace file more flexible, easy to access and edit. The Trace file has the XML-like header, which contains all necessary information of simulator's model configuration. XML-based header allows us to make configuration information more detailed if necessary, by adding additional parameters, without losing compatibility with the older format or corrupting proper file structure. The main trace section, which is located right after the header, describes each relevant cache event encountered during simulation.

Each cache access event is presented as a new line. Parameters of the cache events are encoded by one or two characters, the number of these parameters could easily be extended, depending on the required level of details. The structure of the trace section is presented in the figure below.

```
n:Core_0_dl1 e:m c:2284678 s:15 t:5242991 id:1
n:Core_0_il1 e:m c:2284966 s:440 t:147538 id:0
n:Core_0_dl1 e:m c:2284972 s:17 t:4718575 id:0
n:Core_0_dl1 e:m c:2285013 s:0 t:5243427 id:1
n:Core_0_dl1 e:m c:2285014 s:16 t:5242991 id:1
n:Core_0_dl1 e:m c:2285020 s:1 t:5243427 id:1
n:Core_0_dl1 e:m c:2285026 s:17 t:5242966 id:1
```

**Fig. 2.** An example of a trace section in a trace file from a real benchmark execution

Each line of the trace section consists of a set of records with the following structure: <*name*>:<*value*>. The <*name*> field is comprised of one or two characters, showing the type of the parameter being considered. The <*value*> field in each record can be expressed using either textual or numerical representation. The collection of possible values constitute the trace section dictionary. The trace section dictionary is as follows:

*n* — name of the cache being accessed. Here, we can distinguish between the IL1 (level 1 instruction cache), DL1 (level 1 data cache), and UL2 (unified instruction/data L2 cache)

*e* — type of event (cache miss or cache hit).

*c* — cycle count at the time of event.

*s* — cache set number being accessed.

*t* — tag of the address .

*id* — ID of the thread, which generated the respective cache access.

For example, the first line shown in Figure 2 is interpreted in the following manner. During simulation cycle 2284678, an application running on core 0 with thread context id of 1 performed a cache access to set number 15 of the D-L1 cache. The access resulted in a cache miss and the tag part of the address was 5242991. Essentially, CacheVisor reads the records of this nature from the trace file and performs cache visualization based on this information.

The use of offline approach provides more opportunities for the data analysis of each simulation cycle. Specifically, since all necessary information is stored in a file, visualizer's state can be scrolled both forward and backward, as shown in Figure 3(a), which depicts the graphical user interface (GUI) of CacheVisor. This provides additional opportunities for observing the state of the shared cache at any time. Figure 3(b) shows an example of a shared cache snapshot, as depicted by CacheVisor. Here, different colors are used to distinguish the cache blocks owned by different threads that share the cache.

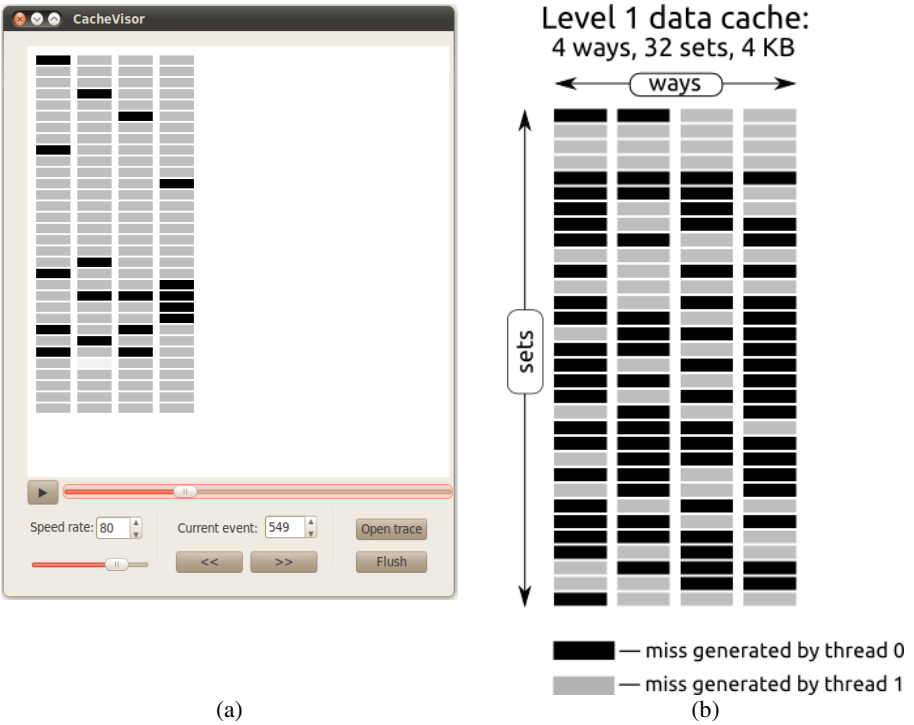



**Fig. 3.** CacheVisor Framework: (a) GUI and (b) an example, showing cache utilization by two threads on Level 1 data cache

## 3 Using CacheVisor in Research and Education

CacheVisor tool exemplifies the application of visualization in computer architecture research. In particular, researchers can utilize CacheVisor (or similar tools) for better understanding the dynamics of shared caches and design better cache sharing algorithms based on this insight. Naturally, CacheVisor can and should be used in undergraduate and graduate-level computer architecture courses to illustrate the course material covering cache memory systems. In fact, our future plans include creating a web interface to enable remote invocation of CacheVisor directly from the web browser. Of course, more detailed information about the memory addresses, cycle counts, and other statistics can be presented in the CacheVisor's window if it is used for educational purposes.

Similar framework can also be applied to the rest of the microprocessor datapath. For example, one can visualize the utilization of various processor queues (reorder buffer, issue queue, load-store queue) to better understand and alleviate resource bottlenecks during the program execution. In addition, it is also possible to visualize power modeling, both within the processor and within the memory subsystem. Our immediate future work involves augmentation of CacheVisor with power modeling and visualization capabilities.

## 4 Concluding Remarks

In this paper we presented CacheVisor - a toolset that supports detailed visualization of modern processor caches. CacheVisor is driven by the results of the cycle-accurate cache simulator, but the visualization engine is well decoupled from the simulator to promote flexibility and portability of the design. CacheVisor can be used to augment the existing simulators in various studies, such as for better understanding of the cache sharing dynamics for shared caches in multithreaded and multicore processors. We also outlined other possible usages of CacheVisor for research as well as education. The public release of CacheVisor is planned in the near future.

**Acknowledgements.** This work is supported by a grant from Ministry of Science and Education of Russian Federation under the Federal Goal-Oriented Program "Scientific and Scientific-Pedagogical Staff of Innovative Russia" for the period of 2009-2013 (by lot 1 "Conducting Research by Teams under the Leadership of Invited Researchers in the Field of Technical and Engineering Sciences", code 2010-1.5-507-001, state contract No 02.740.11.5125).

## References

1. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report No UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
2. From a Few Cores to Many: A Tera-Scale Computing Research Overview (2006), ftp://download.intel.com/rssearch/platform/terascale/terascale_overview_paper.pdf
3. Brooks, D., Bose, P., Schuster, S., Jacobson, H., Kudva, P., Buyuktosunoglu, A., et al.: Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. J. IEEE Micro. 20(6), 26–44 (2000)
4. Brooks, D., Martonosi, M.: Dynamic Thermal Management for High-Performance Microprocessors. In: 7th Int'l Symposium on High-Performance Computer Architecture (HPCA), pp. 171–182. IEEE CS Press, Los Alamitos (2001)
5. M-sim Simulator. Source code and documentation, http://www.cs.binghamton.edu/~msim
6. Casazza, J.: First the Tick, Now the Tock: Intel Microarchitecture (Nahalem). Intel White Paper, http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf
7. Sinharoy, B.: Power 7 Multicore Processor Design. In: 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–1. ACM, New York (2009)
8. Qureshi, M., Patt, Y.: Utility-Based Cache Partitioning: A Low-overhead, High-Performance Runtime Mechanism to Partition Shared Caches. In: 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 423–432. IEEE CS, Washington (2006)
9. Xie, Y., Loh, G.H.: PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In: 36th IEEE/ACM International Symposium on Computer Architecture (ISCA), pp. 174–183. ACM, New York (2009)

# The LuNA Library of Parallel Numerical Fragmented Subroutines*

Sergey Kireev[1,2], Victor Malyshkin[1,2], and Hamido Fujita[3]

[1] ICMMG SB RAS, Novosibirsk, Russia
[2] Novosibirsk State University, Russia
[3] Iwate Prefectural University, Japan
{malysh,kireev}@ssd.sscc.ru

**Abstract.** The LuNA library of parallel numerical fragmented subroutines is now under development. It is aimed at automated treatment of a range of important and stubborn properties of numerical subroutines. The method of algorithm and program fragmentation is used. The library is being developed with the LuNA fragmented programming system. It provides all the necessary properties of subroutines and their high portability.

## 1 Introduction

Libraries of standard subroutines play an important role in the sequential implementation of numerical models. The development of similar libraries of parallel numerical subroutines is faces with serious difficulties. The problems arisen are caused by the necessity to automatically provide the dynamic properties of application parallel subroutines, such as dynamic **tunability** to all the available resources of a computer system in the course of execution: internode data transfers in parallel with the program execution to reduce overhead, dynamic load balancing, etc. Organization of library subroutines and execution of their calls from sequential and/or parallel application programs should be made in such a way that to avoid the necessity to program the dynamic properties.

There is also another range of important library subroutines properties that should be provided, too. Firstly, a subroutine should contain a non-changeable algorithm description. The algorithm representation should not be changed if the library in question is used on another multicomputer. Secondly, all the subroutine transformations, that need to be done to port the library to a new multicomputer, should be concentrated in a system component(s), but not in the text of a subroutine for a better **portability** of the library. Thirdly, the accumulated fund of algorithms and programs should be possible to use in the process of the library creation, especially, the program codes. These properties of the library subroutines will provide the cumulative effect of the subroutine use.

Most of today's parallel libraries do not separate an algorithm description from its implementation in a certain architecture [1,2,3,4,5]. The library subroutines are efficiently operating in a certain multicomputer environment, and a

* The work was supported by RFBR grant 10-07-00454a.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 290–301, 2011.

serious effort is required for porting them to a new one. In a number of projects, Directed Acyclic Graph (DAG) [5,6,7,8] is used for an algorithm representation. In the PLASMA library [5], the DAG representation is used only to represent algorithms with required properties, and is hardcoded into the subroutines. Some efforts to separate an algorithm and its implementation were made in the DPLASMA library [6], which is now still under the development. In the SMP Superscalar programming environment [7], the DAG is dynamically generated in the course of execution and later used to make some runtime scheduling. However, this approach does not provide potentialities for the static program analysis before runtime. Another example is Uintah framework [8], where the DAG representation of a certain algorithm is used for allowing the runtime system to implement some dynamic properties.

Some promising frameworks [9,10] automatically provide a range of dynamic properties of parallel programs. Their drawback is the same: an algorithm is hidden inside the program code that closes potentialities for the static program analysis.

In our effort to develop the LuNA library of parallel numerical subroutines, that satisfies all the above requirements, we are basing on the idea of algorithms and programs fragmentation [11,12], which in turn is based on the method of parallel program synthesis [13]. The idea in question was embodied into the LuNA fragmented programming system [14]. The algorithms and programs fragmentation was exploited in many publications [11,12,14,15]. A range of numerical algorithms were fragmented including such a complex numerical method as Particle-In-Cell (PIC) [15].

## 2 Introduction to Fragmented Programming

The fragmented program (FP) development includes the following stages:

1. algorithm specification,
2. algorithm fragmentation,
3. FP creation: resources allocation and control development for executing an algorithm on a certain hardware.

The fragmented approach to parallel program creation is demonstrated on a matrices multiplication subroutine example.

**Algorithm specification.** The following sequential algorithm of the square matrices multiplication is used:

$$C = AB, \quad A = (a_{i,j})_{i,j=\overline{1,N}}, \quad B = (b_{i,j})_{i,j=\overline{1,N}}, \quad C = (c_{i,j})_{i,j=\overline{1,N}},$$
$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}, \quad i,j,k = \overline{1,N}. \tag{1}$$

**Algorithm fragmentation.** The first step of the fragmented algorithm development is data fragmentation. Most of numerical algorithms have a regular structure of data and computations, which can be naturally divided into parts.

Division of algorithm data into parts is called *data fragmentation*. For example, if the initial algorithm data structure is an N-dimensional array, then it can be fragmented into N-dimensional sub-arrays. So, a fragmented algorithm will process the N-dimensional array of subarrays, whose entries are called *data fragments*.

In the fragmented matrices multiplication algorithm, the square matrices $A$, $B$ and $C$ of $N \times N$ size are divided into submatrices of $M \times M$ size, which are data fragments (Figure 1). For simplicity $K = N/M$ is considered to be an integer. The type of a data fragment of the example is denoted as `matrix`:

`matrix` ≡ `array [1..M, 1..M] of real`.

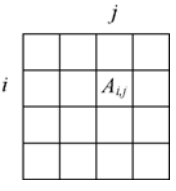



**Fig. 1.** 2D array fragmentation example

So, the set of data fragments is as follows:

$$DF = \left\{A_{i,j} | i,j = \overline{1,K}\right\} \cup \left\{B_{i,j} | i,j = \overline{1,K}\right\} \cup \\ \cup \left\{C_{i,j} | i,j = \overline{1,K}\right\} \cup \left\{D^k_{i,j} | i,j,k = \overline{1,K}\right\},$$

where $A_{i,j}$, $B_{i,j}$, $C_{i,j}$, $D^k_{i,j}$ : `matrix`.

Here $A_{i,j}$, $B_{i,j}$, $C_{i,j}$ and $D^k_{i,j}$ are the names of data fragments. $A_{i,j}$, in particular, denotes a submatrix in the $i$-th row and the $j$-th column of $K \times K$ array of data fragments. In Figure 1, matrix $A$ fragmentation for $K = 4$ is presented. The output data fragments $C_{i,j}$ are computed by the following fragmented algorithm:

$$\begin{aligned} &1)\ D^k_{i,j} = A_{i,k} B_{k,j}, \\ &2)\ C_{i,j} = \sum_{k=1}^{N} D^k_{i,j}. \end{aligned} \tag{2}$$

The product $D^k_{i,j} = A_{i,k} B_{k,j}$ is a partial sum of $C_{i,j}$. The data fragments have the same structure as the initial data structure. Thus, the matrices multiplication can be implemented by a ready-made sequential procedure, for example, the BLAS subroutine SGEMM. Algorithm (2) is defined by a set of functional terms, depicted in Fig. 2. The operations of the functional terms are called *fragments of computation* (FoC).

**The FP creation.** In order to execute the above algorithm on a multicomputer, resources should be assigned to all the data fragments and the FoCs, the order of the FoCs execution should be defined and some optimizations can be done. The FP is a fragmented algorithm supplemented with additional information needed for its execution.

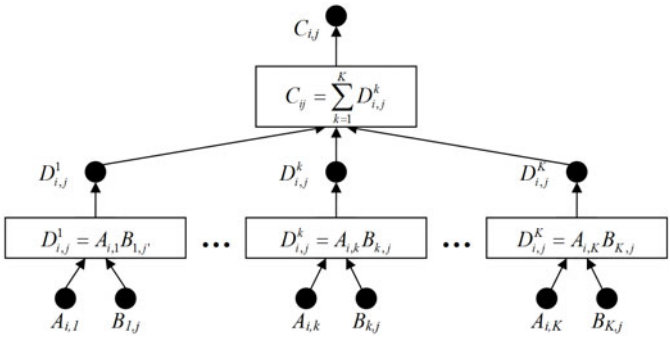



**Fig. 2.** Functional terms defining the fragmented matrices multiplication algorithm

The LuNA fragmented programming system was implemented as a compiler and a runtime system. At the first stage of FP processing, the compiler converts the FP code to the internal form, performing some global optimizations. The compiler makes the static resources assignment for those data fragments and FoCs, for which it may be done, and imposes some restrictions on the order of FoCs execution and further dynamic resources allocation. At the second stage of FP processing, the runtime system executes the operations of the functional terms, defining the fragmented algorithm ([2]). The runtime system dynamically allocates multicomputer resources for the rest of data fragments and FoCs and executes all the FoCs in a certain permissible order that does not contradict the information dependencies between FoCs, imposed by the functional terms of the algorithm definition, and the compiler imposed restrictions.

## 3 The LuNA Subroutines

The FP written in LuNA language does not actually define explicitly a set of functional terms. A fragmented algorithm in LuNA language is defined as:

- DF – a set of data fragments,
- CF – a set of FoCs,
- $\rho$ – partial order relation on a set of FoCs.

The sets of data fragments and FoCs and the partial order relation on a set of FoCs make possible to dynamically construct the functional terms in the course of execution. The order relation $\rho$ includes data dependencies along with other dependencies, added by a compiler at the stage of global optimization and explicitly defined by the programmer. In the course of program execution, each FoC is only once executed, whereas data fragments are multiple assignment variables and denote memory locations. This is done intentionally in order to help the runtime system to construct a high-quality resources allocation.

The LuNA FP is a fragmented algorithm with partially assigned resources. The resources for some of data fragments and FoCs are to be dynamically allocated. Such a program representation does not contain unnecessary dependencies

between FoCs and allows many different ways of program execution. In what follows, several examples of fragmented library subroutines are presented.

### 3.1 Matrices Multiplication

Here the matrices multiplication fragmented algorithm (2) written in LuNA language is presented.

```
df a[i,k] := block(4*M*M) | i=0..K-1, k=0..K-1;
df b[k,j] := block(4*M*M) | k=0..K-1, j=0..K-1;
df c[i,j] := block(4*M*M) | i=0..K-1, j=0..K-1;
df d[i,j,k] := block(4*M*M) | i=0..K-1, j=0..K-1, k=0..K-1;
```

The keyword `df` specifies the data fragments description:

$$DF = \left\{a_{i,k} | i = \overline{0, K-1}, k = \overline{0, K-1}\right\} \cup$$
$$\cup \left\{b_{k,j} | k = \overline{0, K-1}, j = \overline{0, K-1}\right\} \cup \left\{c_{i,j} | i = \overline{0, K-1}, j = \overline{0, K-1}\right\} \cup$$
$$\cup \left\{d_{i,j,k} | i = \overline{0, K-1}, j = \overline{0, K-1}, k = \overline{0, K-1}\right\}.$$

The data fragments $a_{i,k}$ and $b_{k,j}$ are input, the data fragments $c_{i,j}$ are output of the FoC `mul`. For every data fragment the keyword `block` defines its size in bytes, $M$ and $K$ being parameters of the algorithm.

```
cf initc[i,j] := proc_zero<M,M> (out: c[i,j])
  | i=0..K-1, j=0..K-1;
cf mul[i,j,k] := proc_mmul<M,M,M> (in: a[i,k],b[k,j];
  out: d[i,j,k]) | i=0..K-1, j=0..K-1, k=0..K-1;
cf sum[i,j,k] := proc_add<M,M> (in: d[i,j,k],c[i,j]; out: c[i,j])
  | i=0..K-1, j=0..K-1, k=0..K-1;
```

The keyword `cf` specifies a set of FoCs:

$$CF = \left\{initc_{i,j} | i = \overline{0, K-1}, j = \overline{0, K-1}\right\} \cup$$
$$\cup \left\{mul_{i,j,k} | i = \overline{0, K-1}, j = \overline{0, K-1}, k = \overline{0, K-1}\right\} \cup$$
$$\cup \left\{sum_{i,j,k} | i = \overline{0, K-1}, j = \overline{0, K-1}, k = \overline{0, K-1}\right\}.$$

For every FoC the corresponding procedure, implementing it, is declared as well as parameters in the angle brackets. Input and output data fragments are denoted by the keywords `in` and `out`. For example, every FoC `initc[i,j]` for any $i = \overline{0, K-1}$ and $j = \overline{0, K-1}$ is implemented by `proc_zero` procedure and produces one output data fragment `c[i,j]`. The procedures `proc_zero`, `proc_mmul` and `proc_add` are ordinary sequential procedures written in C/C++. The procedure `proc_zero` fills the $M \times M$ matrix with zero values, `proc_mmul` performs $M \times M$ matrices multiplication, and `proc_add` performs $M \times M$ matrices summation.

```
initc[i,j] < sum[i,j,k] | i=0..K-1, j=0..K-1, k=0..K-1;
mul[i,j,k] < sum[i,j,k] | i=0..K-1, j=0..K-1, k=0..K-1;
```

The partial order relation on a set of FoCs is defined as a set of pairs denoted by a terminal symbol "less than". Above, the FP denotes the following order relation:

$$\rho = \left\{(initc_{i,j}, sum_{i,j,k}) | i = \overline{0, K-1}, j = \overline{0, K-1}, k = \overline{0, K-1}\right\} \cup$$
$$\cup \left\{(mul_{i,j,k}, sum_{i,j,k}) | i = \overline{0, K-1}, j = \overline{0, K-1}, k = \overline{0, K-1}\right\}.$$

In every pair, the second FoC must start its execution not earlier than the first one has finished its execution. The whole FP execution is finished when all the started FoCs have been finished and no other FoC may start.

The above-presented LuNA FP is portable. It does not depend on the computer architecture and can already be executed by the runtime system. The partial order relation $\rho$ defines a set of different ways of FP execution, some of them being suitable and some being not. For example, the runtime system may first execute all $mul_{i,j,k}$ and then all $sum_{i,j,k}$, so it would be necessary to hold in memory all $K \times K \times K$ data fragments $d_{i,j,k}$. But if for every pair $i, j$ all FoCs $mul_{i,j,k}$ and $sum_{i,j,k}$ are inserted into the separate sets $S_{i,j}$, then computing all the FoCs from a certain $S_{i,j}$, then all FoCs from another $S_{i1,j1}$, and so on, it would be necessary to hold only $K$ data fragments $d_{i,j,k}$ at any time for each pair $i, j$.

In [14], the means of FP optimization in the LuNA system were presented. They consist in a set of the programmer's recommendations for resources allocation and the order of FoCs execution:

- Definition of the priorities of choosing FoCs for execution.
- Definition of the groups of FoCs with Group Member First (GMF) strategy. If any FoC of such a group started its execution, then all the FoCs within this group should be started before the other FoCs. In the above example of execution of the matrix multiplication program, the sets $S_{i,j}$ are groups with GMF strategy.
- Definition of estimates of the computational complexity of FoCs.
- Definition of the binary neighborhood relation on a set of FoCs as a set of pairs of FoCs. If two FoCs are defined as neighbors, then the runtime system will try to locate them on the same or the neighboring cluster node.

Using the above recommendations, the programmer can provide the runtime system with additional information about a desirable way of FP execution. The recommendations contain no assumptions about the exact number of fragments or resources and maintain FP portability. In addition, the programmer can use the direct instructions in order to assign FoCs to particular processors and/or to include additional elements into $\rho$ in order to reduce the number of different ways of the FP execution.

An appropriate way of execution for the FP of matrices multiplication can be defined, for example, by strengthening the order relation and assigning FoCs to the processors explicitly:

```
initc[i,j].location = i * PE_COUNT / K | i=0..K-1, j=0..K-1;
mul[i,j,k].location = i * PE_COUNT / K
  | i=0..K-1, j=0..K-1, k=0..K-1;
sum[i,j,k].location = i * PE_COUNT / K
  | i=0..K-1, j=0..K-1, k=0..K-1;
mul[i,(i+j)%K,k] < mul[i,(i+j+1)%K,k]
  | i=0..K-1, j=0..K-2, k=0..K-1;
```

Here the parallel matrices multiplication program on a line of computer nodes is defined. The matrix $A$ is distributed among computer nodes by rows, and the distribution is fixed. The matrix $B$ is distributed by columns, and these data fragments are cyclically shifted from one node to another in the course of computation (Fig. 3).

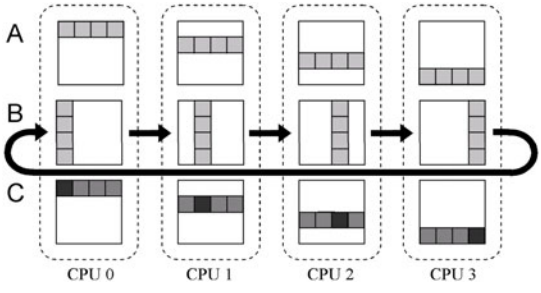



**Fig. 3.** Data fragments distribution for FP of matrices multiplication

### 3.2 LU-Factorization

Factorization of a square matrix $A$ into the product of a lower triangular matrix $L$ and an upper triangular matrix $U$ can be performed using the formulas:

$$l_{i,j} = a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j}, \quad u_{i,j} = \frac{1}{l_{i,i}} \left[ a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j} \right],$$

where $a_{i,j}$, $l_{i,j}$ and $u_{i,j}$ are entries of the corresponding matrices $A$, $L$ and $U$. The LU-factorization fragmented algorithm written in LuNA language is as follows:

```
df A[i,j] := block(4*M*M) | i=0..K-1, j=0..K-1;

cf fd[i] := fd<M> (in: A[i,i]; out: A[i,i]) | i=0..K-1;
cf fl[j,i] := fl<M,M> (in: A[i,i],A[j,i]; out: A[j,i])
  | i=0..K-1, j=i+1..K-1;
cf fu[i,k] := fu<M,M> (in: A[i,i],A[i,k]; out: A[i,k])
  | i=0..K-1, k=i+1..K-1;
cf fg[i,j,k] := fg<M,M,M> (in: A[j,i],A[i,k],A[j,k];
  out: A[j,k]) | i=0..K-1, j=i+1..K-1, k=i+1..K-1;
```

```
fd[i] < fl[j,i] | i=0..K-1, j=i+1..K-1;
fd[i] < fu[i,k] | i=0..K-1, k=i+1..K-1;
fl[j,i] < fg[i,j,k] | i=0..K-1, j=i+1..K-1, k=i+1..K-1;
fu[i,k] < fg[i,j,k] | i=0..K-1, j=i+1..K-1, k=i+1..K-1;
fg[i,i+1,i+1] < fd[i+1] | i=0..K-2;
fg[i-1,j,i]   < fl[j,i] | i=1..K-2, j=i+1..K-1;
fg[i-1,i,k]   < fu[i,k] | i=1..K-2, k=i+1..K-1;
fg[i-1,j,k] < fg[i,j,k] | i=1..K-2, j=i+1..K-1, k=i+1..K-1;
```

The algorithm fragmentation is made in such a way in order to be able to change the fragment size $M$ up to 1 entry. The data fragments $a_{i,j}$ are input and output for the FP, whereas a result obtained is written down into the initial matrix location. The order of FoCs execution is depicted in Figure 4. All FoCs can be divided into subsets by the index $i$. The first subset contains all FoCs with $i = 1$ (marked with $1, 2, \ldots$ in Figure 4a), the second one contains all FoCs with $i = 2$ (marked with $2, 3, \ldots$ in Figure 4a), and so on. Dependencies between FoCs inside each subset are illustrated in Figure 4b. In addition, each FoC with the indices $i, j, k$ $(i > 1)$ depends on the corresponding FoC with the indices $i - 1, j, k$.

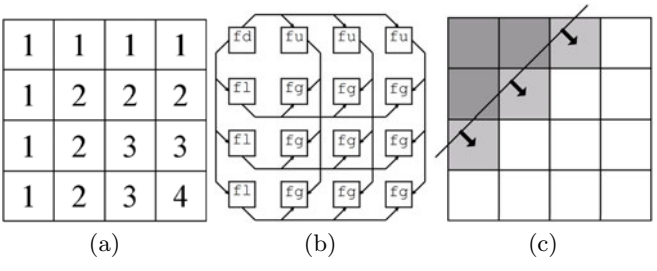



**Fig. 4.** The order of FoCs execution for LU-factorization fragmented algorithm

An example of inefficient order of the FoCs execution is a sequential execution of the above-mentioned subsets for $i = 1, 2, 3, \ldots$. The FoCs from each subset depend on one corresponding FoC $FD_i$, so there would be a bottleneck for each value $i$. The better way is to organize computations in a wave-like manner. The wave of execution starts from $i = j = k = 1$ and is uniformly spread in all the directions (Figure 4c). The following LuNA-code sets a proper way of execution using the priorities:

```
fd[i].priority = 2*K          | i=0..K-1;
fl[j,i].priority = 2*K-j-i    | i=0..K-1, j=i+1..K-1;
fu[i,k].priority = 2*K-i-k    | i=0..K-1, k=i+1..K-1;
fg[i,j,k].priority = 2*K-j-k | i=0..K-1, j=i+1..K-1, k=i+1..K-1;
```

### 3.3 Computations on Regular Meshes

A wide class of numerical simulation problems are solved using iterative algorithms on multidimensional regular meshes. Such a computation is organized as a sequence of iterations, with all mesh nodes being processed. The order, in which the mesh nodes are processed, varies from one application problem to another. For example, for a certain problem on a rectangular mesh, there may be dependencies between the mesh nodes along some axes and no dependencies along the others. As a rule, the computation assigned to a given mesh node depends on values in the neighboring mesh nodes from several preceding iterations.

An obvious way of an N-dimensional mesh fragmentation is its division into N-dimensional submeshes of an equal size with intersections. The submeshes are processed in the same order as entries of the initial mesh. In addition, an exchange of the fragment borders should take place between the neighboring mesh fragments after each iteration. The exchange is normally organized by the intersection data fragments.

The algorithm below performs $T$ steps of iterative computations on a 2D regular $N \times N$ mesh, which is defined as an array of data fragments $X$. For simplicity of presentation, the mesh decomposition is made along one axis dividing the initial mesh into $K$ $N \times M$ layers, where $M = N/K$ (Fig. 5). Such a layer is taken as a data fragment; $X1$ and $X2$ are data fragments to hold the border values. Such an algorithm may be used, for example, for a 2D Poisson equation solution by the Jacobi method.

```
const M = N / K;
const B = M+2;

df X [i,it] := block(8*N*B) | i=0..K-1, it=0..T;
df X1[i,it] := block(8*N)   | i=1..K-1, it=0..T-1;
df X2[i,it] := block(8*N)   | i=0..K-2, it=0..T-1;

cf step[i,it] := step<B,N,it>(in: X[i,it],F[i];
  out: X[i,it+1],DX[i,it]) | i=0..K-1, it=0..T-1;
cf getb[i,it] := getbud<B,N,i,it>(in: X[i,it];
  out: X1[i,it],X2[i,it]) | i=1..K-2, it=1..T-1;
cf getb[i,it] := getbu <B,N>(in: X[i,it]; out: X2[i,it])
  | i=0, it=1..T-1;
cf getb[i,it] := getbd <B,N>(in: X[i,it]; out: X1[i,it])
  | i=K-1, it=1..T-1;
cf setb[i,it] := setbud<B,N >(in: X[i,it],X2[i-1,it],X1[i+1,it];
  out: X[i,it]) | i=1..K-2, it=1..T-1;
cf setb[i,it] := setbu<B,N>(in: X[i,it],X1[i+1,it];
  out: X[i,it]) | i=0, it=1..T-1;
cf setb[i,it] := setbd<B,N>(in: X[i,it],X2[i-1,it];
  out: X[i,it]) | i=K-1, it=1..T-1;
```

```
step[i,it-1] < getb[i,it] | i=0..K-1, it=1..T-1;
step[i,it-1] < setb[i,it] | i=0..K-1, it=1..T-1;
getb[i-1,it] < setb[i,it] | i=1..K-1, it=1..T-1;
getb[i+1,it] < setb[i,it] | i=0..K-2, it=1..T-1;
setb[i,it] < step[i,it] | i=0..K-1, it=1..T-1;
```

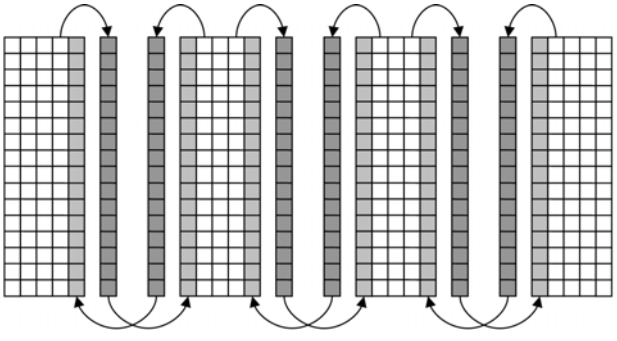


**Fig. 5.** An example of mesh decomposition for $K = 4$

To help the runtime system to efficiently execute the algorithm, the programmer can provide such information as a neighborhood relation on a set of FoCs:

```
neighbors step[i,it], step[i+1,it] | i=0..K-2, it=0..T-1;
```

With such information, the runtime system will try to locate the neighboring FoCs to the same or the neighboring computer nodes, distributing resources and computations more efficiently.

## 4 Concluding Remarks

The fragmented approach to library numerical subroutines creation has been presented. Examples of FPs that do not depend on the architecture of a multicomputer are shown. A fragmented subroutine contains only the unchangeable algorithm description. This allows the programmer not to pay attention to the low-level implementation details and to program on a high level.

**The algorithm fragmentation methodology.** The methodology of the algorithm fragmentation is still under development. However, the above examples of the algorithm fragmentation allow us to state the following FP properties that the programmer should strive to achieve:

- The initial data structure should be divided into data fragments that have the same structure - a matrix should be divided into submatrices, a mesh - into submeshes, a tree - into subtrees, etc. It will make possible to use a ready-made sequential code or optimized library subroutines to implement FoCs.

- All the data fragments should have approximately the same size, and all the FoCs should have approximately the same computational complexity for both automatic and efficient resources allocation and dynamic load balancing.
- The size of data fragments and computational complexity of FoCs should be a parameter of a fragmented algorithm and should not depend on the task size. In such a way, only the number of fragments will be changed when the task size is changed and the same algorithms of control and resources allocation can be used.
- The data fragments should form the same structure as initial data structure, and the order of FoCs execution should be the same as the order of operations in the initial algorithm. This property will allow changing the fragment size (up to 1 entry).

The numerical algorithm fragmentation is not an easy task and often requires an essential transformation of the initial algorithm [16].

The unified FP representation as sets of data fragments, FoCs and the order $\rho$ allows one to efficiently implement a runtime system, which automatically provides all the dynamical program properties. The FP representation has no unnecessary dependencies among FoCs and allows many ways of their execution, from which the LuNA system tries to choose the best one. In contrast to the other parallel programming environments, the LuNA compiler analyzes the whole program and makes all decisions that can be statically taken. The runtime system, in its turn, makes the decisions dynamically, providing the dynamical properties of a FP.

**Future work.** In the future, we are planning to accumulate and to analyze the experience gained in the numerical algorithms fragmentation and FP development in order to create a unified methodology of numerical algorithms fragmentation. Currently, the LuNA library contains most of BLAS fragmented subroutines. The fragmentation of several LAPACK subroutines and operations with sparse matrices is now under development. Our future plans are to include into the library a set of essential components and frames for implementation of widely used numerical methods, such as PIC-method, and to apply them for implementation of a number of large-scale numerical problems.

## References

1. Intel MKL library, http://software.intel.com/en-us/articles/intel-mkl/
2. NAG Parallel library, http://www.nag.com/numeric/numerical_libraries.asp
3. PETSc library, http://www.mcs.anl.gov/petsc/petsc-as/
4. NIST Sparse BLAS library, http://math.nist.gov/spblas/
5. PLASMA Library, http://icl.cs.utk.edu/plasma/
6. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, H., Herault, T., Kurzak, J., Langou, J., Lemariner, P., Ltaief, H., Luszczek, P., YarKhan, A., Dongarra, J.: Distibuted Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures: DPLASMA. University of Tennessee Computer Science Technical Report, UT-CS-10-660, September 15 (2010)

7. SMP Superscalar, http://www.bsc.es/smpsuperscalar
8. Berzins, M., Luitjens, J., Meng, Q., Harman, T., Wight, C.A., Peterson, J.R.: Uintah - A Scalable Framework for Hazard Analysis. In: Proceedings of the Teragrid 2010 Conference, vol. (3) (2010)
9. Charm++, http://charm.cs.uiuc.edu/
10. ProActive, http://proactive.inria.fr/
11. Malyshkin, V.E., Sorokin, S.B., Chajuk, K.G.: Fragmentation of numerical algorithms for the parallel subroutines library. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 331–343. Springer, Heidelberg (2009)
12. Kireev, S., Malyshkin, V.: Fragmentation of Numerical Algorithms for Parallel Subroutines Library. The Journal of Supercomputing 58(1) (2011)
13. Valkovskii V., Malyshkin V.: Parallel Program Synthesis on the Basis of Computational Models. Novosibirsk, Nauka (1988) (in Russian. Sintez Parallel'nykh Program i System na Vychislitel'nykh Modelyakh)
14. Malyshkin, V.E., Perepelkin, V.A.: Optimization of parallel execution of numerical programs in luNA fragmented programming system. In: Hsu, C.-H., Malyshkin, V. (eds.) MTPP 2010. LNCS, vol. 6083, pp. 1–10. Springer, Heidelberg (2010)
15. Kraeva, M.A., Malyshkin, V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. The Int. Journal on Future Generation Computer Systems 17(6), 755–765 (2001)
16. Terekhov, A.V.: Parallel Dichotomy Algorithm for solving tridiagonal system of linear equations with multiple right-hand sides. Parallel Computing 36(8), 423–438 (2010)

# PARMONC - A Software Library for Massively Parallel Stochastic Simulation*

Mikhail Marchenko

The Institute of Computational Mathematics and Mathematical Geophysics SB RAS,
Prospekt Lavrentieva 6, 630090, Novosibirsk, Russia,
Novosibirsk State University,
Ul. Pirogova 2, 630090, Novosibirsk, Russia
Tel.: (383)330-77-21, Fax: (383) 330-87-83
mam@osmf.sscc.ru

**Abstract.** In this paper, the software library PARMONC that was developed for the massively parallel simulation by Monte Carlo method on supercomputers is presented. The "core" of the library is a well tested, fast and reliable long-period parallel random numbers generator. Routines from the PARMONC can be called in the user-supplied programs written in C, C++ or in FORTRAN without explicit usage of MPI instructions. Routines from the PARMONC automatically calculate sample means of interest and the corresponding computation errors. A computational load is automatically distributed among processors in an optimal way. The routines enable resuming the simulation that was previously performed and automatically take into account its results. The PARMONC is implemented on high-performance clusters of the Siberian Supercomputer Center.

**Keywords:** Monte Carlo method, distributed stochastic simulation, random number generator, parallel computation, supercomputers.

## 1 Introduction

It becomes a common point of view that probabilistic imitation models and Monte Carlo method (stochastic simulation) will be widely used for computer-aided simulation in the nearest future. There are a few reasons for such a prediction. First of all, the use of probabilistic models is an adequate way to simulate physical, chemical or biologic phenomena from "first principles". On the other hand, Monte Carlo methods, which realize probabilistic models, can be effectively parallelized in the form of distributed computing. Therefore, a progress in development and implementation of powerful supercomputers gives way to a wide use of Monte Carlo method as a principal instrument for the computer-aided simulation in many scientific areas (see, e.g., [1], [2]).

The main objective of this paper is to introduce PARMONC - a library of easy-to-use programs that was implemented on high-performance clusters of the

* This work was supported by the RFBR grants No 09-01-00639 and No 09-01-00035.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 302–316, 2011.

Siberian Supercomputer Center (http://www2.sscc.ru) and can also be used in other supercomputer centers [3], [4].

The development of the PARMONC (an acronym for PARallel MONte Carlo) is based on the library MONC that was implemented for a network of personal computers [5]. The MONC was intensively used in the Department of Stochastic Simulation in Physics of the Institute of Computational Mathematics and Mathematical Geophysics of the SB RAS in Novosibirsk for a wide area of applications. Also, the MONC was actively applied in the Laboratory of Probability-Theoretical Methods of the Omsk Branch of the Sobolev Institute of Mathematics of the SB RAS to solve various problems in the population biology.

The main objectives of the library development are as follows:

- creation of a software tool suitable for the massively parallel stochastic simulation for a wide range of applications,
- creation of an easy-to-use software framework to parallelize stochastic simulation to be applied without knowledge of MPI language.

There are a number of publications and internet resources dedicated to the parallel random numbers generation (see, e.g., [6], [7], [8]). In comparison to other parallel random number generators (RNGs), the one used for the PARMONC is fairly fast, reliable and has an extremely long period. Also, this parallel RNG is governed by a few parameters defined by the user. Using this generator, it appears possible to scale the stochastic simulation to sufficiently large (practically infinite) number of processors (see Sections 2.4 and 3.5).

A number of publications are devoted to the development of software packages for parallel computations with Monte Carlo method (see, e.g., [9], [10], [11]). Different hardware and software platforms are reported in these publications. In our opinion, the following features distinguish the PARMONC from other software tools and make it an easy-to-use instrument for specialists in the field of stochastic simulation:

- The only thing the user has to do in order to parallelize stochastic simulation is to write in C, C++ or in FORTRAN a sequential subroutine to simulate a single realization of a random object of interest and to pass its name to the PARMONC routines (see Sections 2.3, 3.2 and 4).
- In his/her sequential code, he/she uses a PARMONC function, which implements a parallel RNG, in a usual and convenient way (see Sections 2.3, 2.4 and 3.3).
- In the course of simulation, the PARMONC periodically calculates and saves in files the subtotal results of simulation and the corresponding computation errors (see Sections 2.2, 3.2 and 4).
- The PARMONC provides an easy-to-use technique to resume stochastic simulation after its termination with automatic averaging of the results of the previous simulation (see Sections 3.2 and 4).

## 2 Background

### 2.1 Estimators of Interest in Stochastic Simulation

Initially, Monte Carlo method (or stochastic simulation) was developed to solve problems of radiation transfer. In the last half of the 20-th century, the area of its applications became much wider. Theory of stochastic representations for solutions to equations of mathematical physics was developed. Using the theory, the corresponding numerical stochastic estimators were drawn up. Efficient algorithms of Monte Carlo method were developed in statistical physics (the Metropolis method, the Ising model, e.g.), in the physical and chemical kinetics (modeling multi-particle problems, solving the Boltzmann and Smoluchowski's equations, modeling the chemical reactions and phase transitions, etc.), the queuing theory, financial mathematics, turbulence theory, mathematical biology, etc.

The stochastic simulation is thought to be numerical realization of stochastic representation of a certain object in order to estimate its desired integral features with the use of a law of large numbers [12], [13]. We assume that a functional of interest $\varphi \in R$ is represented as expectation of some random variable $\zeta$:

$$\varphi \approx \mathrm{E}\zeta,$$

provided that its variance $\mathrm{Var}\zeta$ is finite. In this case, one can evaluate the value of $\mathrm{E}\zeta$ using a sample mean:

$$\mathrm{E}\zeta \approx \bar{\zeta} = L^{-1} \sum_{i=1}^{L} \zeta_i \tag{1}$$

where $\zeta_i$ are independent realizations of a random variable $\zeta$. The value of $\bar{\zeta}$ is called a **stochastic estimator for** $\varphi$.

One also needs to evaluate the second moment $\mathrm{E}\zeta^2$ of the random variable

$$\mathrm{E}\zeta^2 \approx \bar{\xi} = L^{-1} \sum_{i=1}^{L} \zeta_i^2$$

in order to estimate variance of a random variable $\zeta$ and its standard deviation

$$\mathrm{Var}\zeta \approx \bar{\sigma}^2 = \bar{\xi} - \bar{\zeta}^2, \ (\mathrm{Var}\zeta)^{0.5} \approx \bar{\sigma}.$$

In Monte Carlo methods [12], [13], a complex random variable is represented as a function

$$\zeta = \zeta(\alpha_1, \alpha_2, \ldots, \alpha_k), \tag{2}$$

where $\alpha_1, \alpha_2, \ldots, \alpha_k$ are independent random variables called **base random numbers** which have uniform distribution on the interval $(0, 1)$. A sequence of the random numbers $\{\alpha_k\}$ is generated with the help of some deterministic numerical algorithm called a **random number generator (RNG)**. Usually, iterative formulas are used [12], [13]:

$$\alpha_{k+1} = f(\alpha_k), k = 0, 1, 2, \ldots$$

where $\alpha_0$ is a fixed quantity.

Thus, calculating the realizations $\zeta_i, i = 1, 2, \ldots, L$ we in turn take base random numbers from the RNG. So, to calculate the sample mean we need a finite set of independent random numbers $R = \{\alpha_1, \alpha_2, \ldots, \alpha_S\}$. We call a **stochastic experiment** the process of calculating the sample mean $\bar{\zeta}$ using a particular set of base random numbers $R$. Usually, $R$ is a subsequence of the general sequence $\{\alpha_k\}$ of base random numbers. Using a different set $R' = \{\alpha'_1, \alpha'_2, \ldots, \alpha'_{S'}\}$ (or a different subsequence) of the base random numbers that are independent of the base random numbers from $R$, we finally obtain an independent value of the sample mean. In other words, we carry out the stochastic experiment which is independent of the first one.

A confidence interval of the confidence level $\lambda$ for the expectation $\mathrm{E}\zeta$ is defined by the formula

$$\lambda = \mathrm{P}(|\bar{\zeta} - \mathrm{E}\zeta| \le \gamma(\lambda)(\mathrm{Var}\zeta)^{0.5}L^{-0.5}) \approx \mathrm{P}(|\bar{\zeta} - \mathrm{E}\zeta| \le \gamma(\lambda)\bar{\sigma}L^{-0.5}). \quad (3)$$

According to Tables of a standard normal distribution, $\gamma(\lambda) = 3$ for $\lambda = 0.997$. A value of an **absolute (stochastic) error** $\bar{\varepsilon}$ of the stochastic estimator $\bar{\zeta}$ is given by the formula

$$\bar{\varepsilon} = 3(\mathrm{Var}\zeta)^{0.5}L^{-0.5} \approx 3\bar{\sigma}L^{-0.5}$$

and the value of a **relative (stochastic) error** is given by the formula

$$\bar{\rho} = \bar{\varepsilon}/\bar{\zeta} \cdot 100\%.$$

Let us extend the conception of realization of a random object. Assume that at the same time the simulation gives different independent values. It is convenient to represent them as a matrix $[\zeta_{ij}], 1 \le i \le n_1, 1 \le j \le n_2$. We will also call it a realization (a realization of a random object). After averaging, the following matrices are automatically calculated in the PARMONC:

- $[\bar{\zeta}_{ij}]$ – a matrix of the sample means,
- $[\bar{\varepsilon}_{ij}]$ – a corresponding matrix of the absolute errors,
- $[\bar{\rho}_{ij}]$ – a corresponding matrix of the relative errors,
- $[\bar{\sigma}^2_{ij}]$ – a corresponding matrix of the sample variances.

Also, the following values are automatically calculated: $\bar{\varepsilon}_{max} = \max_{i,j} \bar{\varepsilon}_{ij}$ is the upper bound for the entries of the matrix of the absolute errors; $\bar{\rho}_{max} = \max_{i,j} \bar{\rho}_{ij}$ is the upper bound for the entries of the matrix of the relative errors; $\bar{\sigma}^2_{max} = \max_{i,j} \bar{\sigma}^2_{ij}$ is the upper bound for the entries of the matrix of the sample variances.

In many applications, the above-mentioned matrices and values give an exhaustive information about the stochastic simulation.

### 2.2 Parallelization of Stochastic Simulation

A problem arises when a **computational cost** of the estimator (or computational expenses for obtaining a desired level of the absolute/relative error) is too large. On the average, the computational cost is proportional to the value

$$C(\zeta) = \tau_\zeta \mathrm{Var}\zeta,$$

where $\tau_\zeta$ is a mean computer time to simulate a single realization of $\zeta$. Also, it is clear from formula (3) that the sample volume $L$ needed for obtaining a desired level of accuracy is proportional to the variance $\mathrm{Var}\zeta$.

To decrease the computational cost, the simulation of statistically independent realizations may be distributed among $M$ processors (numbered from 0 to $M-1$). At some moment all the processors send subtotal sample means to a dedicated processor (e.g., to 0-th), and the parallel modification of the estimator is given by the formula

$$\bar{\zeta}_M = (\sum_{m=0}^{M-1} l_m)^{-1} \sum_{m=0}^{M-1} l_m \bar{\zeta}^{(m)}, \tag{4}$$

where $l_m$ is a sample volume corresponding to the $m$-th processor, $\bar{\zeta}^{(m)}$ is a corresponding sample mean.

For the massively parallel stochastic simulation, the necessary quantity of base random numbers is very large, and the choice of a parallel RNG must be made with care. For example, a period of a well known RNG with special parameters $r = 40$ and $A = 5^{17}$ is equal to $2^{38} \approx 2.75 \cdot 10^{11}$ (see formulas (6) and (7)) [12], [13]. Such a period is not sufficient for the up-to-date computations: the simulation of a single realization may demand a quantity of base random numbers comparable with the whole period of this generator [5].

Therefore, requirements for a parallel RNG are very rigorous. An important requirement is that sequences of base random numbers $\{\alpha_k\}$ generated on different processors must be independent of each other. Also, base random numbers produced on different processors must have good statistical properties. A necessary information about this subject may be found in [5]. In case of a "good" generator, with increasing the number of processors $M$ and the total sample volume $\sum_{m=0}^{M-1} l_m$, the value of parallel modification (4) goes to $\mathrm{E}\zeta$. Naturally, the simulation of realizations must be effectively performed on processors without any problems as those related to memory limitations, etc.

According to this parallelization technique, the stochastic simulation of realizations on different processors is performed in asynchronous mode. It is clear that it is possible to neglect the time expenses for quite rare data exchanges between the 0-th processor and the other ones. In this case, the variance $\mathrm{Var}\zeta$ remains the same but the value of $\tau_\zeta$ is decreased. As a result, the value of $\tau_\zeta$ (and, respectively, the value of $C(\zeta)$) is decreased by $M$ times thus giving the optimal parallelization [5].

It is possible to exchange data at the end of simulation when all the processors have simulated the dedicated number of realizations. However, it is not advisable for several reasons. First of all, it is desirable to control the absolute and relative stochastic errors during the simulation. On the other hand, it is useful to create periodic "save-points" of the simulation. For this reason, we modify the parallelization technique in the following way.

Let the $m$-th processor ($m = 0, 1, \ldots, M-1$) periodically sends entries of the matrices $[\bar{\zeta}_{ij}^{(m)}]$ and $[\bar{\xi}_{ij}^{(m)}]$ and the corresponding sample volume $l_m$ (calculated

by the moment of sending data) to the 0-th processor. In turn, the 0-th processor periodically receives all the sums $\{\bar{\zeta}_{ij}^{(m)}\}$, $\{\bar{\xi}_{ij}^{(m)}\}$ and the sample volumes $\{l_m\}$, $m = 0, 1, \ldots, M-1$, that were sent to it. Then the 0-th processor averages the sample moments:

$$\bar{\zeta}_{ij} = l^{-1} \sum_{m=0}^{M-1} l_m \bar{\zeta}_{ij}^{(m)}, \ \bar{\xi}_{ij} = l^{-1} \sum_{m=0}^{M-1} l_m \bar{\xi}_{ij}^{(m)}, \tag{5}$$

where $l = \sum_{m=0}^{M-1} l_m$, calculates the sample variances $\bar{\sigma}_{ij}^2$, the absolute $\bar{\varepsilon}_{ij}$ and the relative $\bar{\rho}_{ij}$ errors of the estimators $\bar{\zeta}_{ij}$. Then the 0-th processor saves the matrices $[\bar{\zeta}_{ij}], [\bar{\varepsilon}_{ij}], [\bar{\rho}_{ij}]$ and $[\bar{\sigma}_{ij}^2]$ in files. Note that the sample volumes $l_m$, $m = 0, 1, \ldots, M-1$ may be different at the moment of passing data. A reason for this fact may consist in different performances of processors or in the diversity of time expenses for the computation of different realizations.

If the frequency of the data exchange with the 0-th processor is not very high, we can neglect the time expenses for the periodical data exchange and averaging. Therefore, the modified parallelization technique enables us to reduce the computational cost of the stochastic simulation nearly by $M$ times. There is also no need to use any load balancing techniques because all the processors work independently and make data exchange in asynchronous mode. This conclusion is proved by an example presented in Subsection 4.

### 2.3 Implementation of Stochastic Simulation

For simplicity let us consider a problem of evaluation of the expectation $\mathrm{E}\zeta$ of a scalar random variable $\zeta$ using the sample mean (1). A typical sequential code (written in C) consists of the following operations:

```
int i, L;
double s, t=0.0;
for(i=0;i<L;i++){
   realization(s);
   t=t+s;
}
t=t/(double)L;
```

Here the argument **L** is the number of independent realizations; **realization** is the name of a sequential routine, which computes a single realization of the random variable $\zeta$ and returns its value to the argument **s**. Finally, the variable **t** gives the value of the sample mean. In the routine **realization** the user calls a function which implements a RNG. The usual use of this function (named **rng()**, e.g.) is as follows:

```
a = rng();
```

Here **a** is the base random number which has the uniform distribution on the interval $(0, 1)$. These numbers are used to simulate necessary complex distributions by formula (2). Given statistically independent outputs from the function **rng()**, all the return values **s** from the subroutine **realization** are statistically independent.

Thereby, a routine, which computes a single realization of a random object, takes the return values from a function that implements a RNG and returns a single realization of a random object. This routine and the specifications for the random object realization are provided by the user. The routine that implements a RNG is considered to be an external routine. In Fig.1 we explain the relationship between main program and data elements in the stochastic simulation.

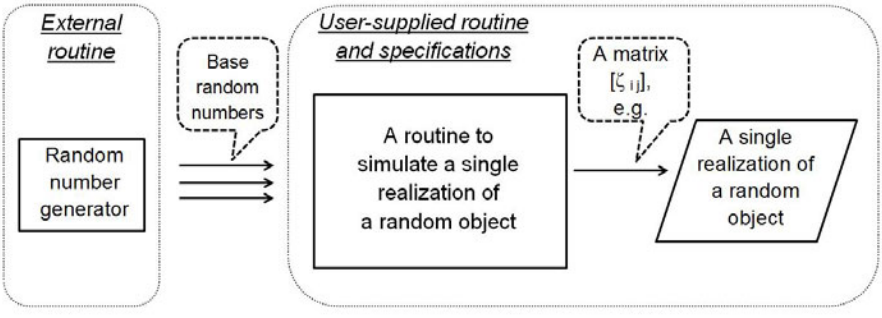



**Fig. 1.** A diagram showing the relationship between the main program and data elements in stochastic simulation

To implement the above-mentioned parallelization technique, the most convenient way is to use a user-defined routine that computes a single realization of a random object as the major piece of the code to be launched on different processors (see Fig. 1). Like in a sequential code, each copy of the routine takes return values from a function that implements the parallel RNG and returns a single realization of a random object. The outputs from all the copies of the user-defined routine (realizations) are taken into account in the course of averaging with the use of formulas (5). This approach is very convenient for specialists in the stochastic simulation because it takes them minimal efforts to adapt their sequential programs for using the PARMONC.

### 2.4 A Parallel RNG

The following base linear congruential generator [12], [13] is used to produce a **general sequence of base random numbers** $\{\alpha_k\}$:

$$u_0 = 1,\ u_{k+1} \equiv u_k A \pmod{2^r},\ \alpha_k = u_k 2^{-r}, \qquad k = 0, 1, 2, \ldots. \tag{6}$$

A period of the congruential generator is

$$L_P = 2^{r-2}. \tag{7}$$

We use the following parameters for the generator [14]:

$$r = 128,\ A \equiv 5^{100109} \pmod{2^{128}}.$$

Therefore, the period of this generator is $2^{126} \approx 10^{38}$. But it is recommended to use the first half of the period only, particularly, the first $2^{125}$ random numbers [12], [13].

In order to obtain independent streams of the base random numbers the general sequence $\{\alpha_k\}$ is divided into subsequences of length $n$ that start with the initial numbers $\tilde{\alpha}_m = \alpha_{nm},\ m = 0, 1, \ldots$. To be exact, the "leaps" of length $n$ are made. **Initial numbers of the subsequences** $\{\tilde{\alpha}_m\}$ are calculated by the formula

$$\tilde{u}_0 = 1,\ \tilde{u}_{m+1} = \tilde{u}_m A(n) \pmod{2^r},\ \tilde{\alpha}_m = \tilde{u}_m\, 2^{-r}, \qquad m = 0, 1, 2, \ldots \tag{8}$$

The multiplier $A(n)$ in this auxiliary generator of the "leaps" of length $n$ is calculated as follows:

$$A(n) \equiv A^n \pmod{2^r}.$$

This parallel generator enables the convergence of the parallel modification (4) to E$\zeta$. It is implemented as a well tested, fast and reliable routine in the Department of Stochastic Simulation in Physics of the Institute of Computational Mathematics and Mathematical Geophysics in Novosibirsk [15], [16]. It was verified on parallel processors using rigorous statistical testing and solving various problems with known solutions. Therefore, using the parallel generator, we may be sure in the correct stochastic simulation on parallel processors.

The PARMONC parallelization technique is to define a hierarchy of embedded subsequences of the general sequence $\{\alpha_k\}$. The PARMONC assigns subsequences of base random numbers to: a) different stochastic experiments, b) different processors and c) different realizations. The technique is as follows:

- within the general sequence $\{\alpha_k\}$, the "leaps" of length $n_e$ are made using (8) in order to define the initial numbers of subsequences that will be used to perform stochastic experiments (when doing it, **"experiments" subsequences** are produced),
- within each "experiment" subsequence, the "leaps" of length $n_p < n_e$ are made using (8) to define the initial numbers of embedded subsequences that will be used on different processors (when doing it, **"processors" subsequences** are produced),
- within each "processor" subsequence, the "leaps" of length $n_r < n_p$ are made using (8) to define the initial numbers of embedded subsequences that will be used to simulate independent realizations (when doing it **"realizations" subsequences** are produced).

So, the hierarchy of the embedded subsequences is as follows:

general sequence $\supset$ "experiments" subsequences

"experiments" subsequence $\supset$ "processors" subsequences

"processors" subsequence $\supset$ "realizations" subsequences

The initialization of a parallel RNG is as follows: the "experiment" subsequence number is defined by the user with the corresponding argument of the subroutine **parmoncf/parmoncc**; the "processor" subsequence number is automatically defined by the PARMONC with a parallel branch number provided by MPI; the "realizations" subsequence number is automatically defined by the PARMONC before starting the simulation of a realization.

The default lengths of "leaps" are as follows:

- $n_e = 2^{115} \approx 10^{34}$ - for "experiments" subsequences,
- $n_p = 2^{98} \approx 10^{29}$ - for "processors" subsequences,
- $n_r = 2^{43} \approx 10^{13}$ - for "realizations" subsequences.

One can therefore perform approximately $2^{125} \cdot 2^{-115} = 2^{10} \approx 10^3$ stochastic experiments; within a single experiment one can use $2^{115} \cdot 2^{-98} = 2^{17} \approx 10^5$ processors at most and on a processor one can simulate $2^{98} \cdot 2^{-43} = 2^{55} \approx 10^{16}$ independent realizations at most.

In the PARMONC the corresponding generator multipliers $A(n_e), A(n_p)$ and $A(n_r)$ are defined to use by default. Nevertheless, one can redefine the default values of $A(n_e), A(n_p)$ and $A(n_r)$ with the use of the command **genparam** (see Subsection 3.5).

## 3 Overview of the Library PARMONC

A description of the PARMONC can be found on the web site of the Siberian Supercomputer Center [3]; the full description is provided in [4].

### 3.1 Contents of the Library

Contents of the PARMONC is as follows:

- **rnd128** - a function to produce a single base random number,
- **parmoncf** - the main subroutine to perform parallel stochastic simulation (for programs written in FORTRAN),
- **parmoncc** - the main subroutine to perform parallel stochastic simulation (for programs written in C),
- **manaver** - a program to average subtotal sample moments calculated on processors (in a manual mode),
- **genparam** - a program to calculate multipliers of the parallel RNG (in a manual mode).

Here **rnd128, parmoncf and parmoncc** are library routines to use in FORTRAN or C/C++ user-supplied programs, **genparam** and **manaver** are executable files to run from a command line. Object files for the library routines are archived to a static library **libparmonc.a**.

The PARMONC software realization does not use any unique features of a specific FORTRAN compiler or a specific MPI implementation. Therefore, it can be compiled and built with any FORTRAN compiler and MPI library and ported to different high-performance clusters or powerful personal computers with multi-core processors.

### 3.2 Subroutines 'parmoncf' and 'parmoncc'

The subroutine **parmoncf/parmoncc** initializes the parallel RNG, distributes the simulation of independent realizations among processors, makes all operations to pass, to collect and to average data and to save the simulation results in files. The simulation results are stored in several files in a special subdirectory of the user's working directory (see Subsection 3.6).

These subroutines take a name of a user-defined routine which computes a single realization of a random object as argument. The main user-supplied program where a call to **parmoncf/parmoncc** is located is considered as a MPI program despite the fact that it does not contain any MPI instructions (see an example in Section 4). This means that it must be compiled, linked and launched according to specific rules determined by a particular MPI realization on the computer.

The argument **res** is a resumption flag. It defines whether the present simulation resumes the previous one or not:

- **res** = 0 in case of a new simulation. In this case the **parmonc** creates brand new files with results.
- **res** = 1 in case of resuming the previous simulation. In this case the **parmonc** automatically takes into account results of the previous simulation (from the corresponding files) and averages it by formulas (5).

The argument **seqnum** is the "experiments" subsequence number (it is equal to 0,1,2, ...). In case of resuming the previous simulation, this argument must be different from the same argument of the previous use of **parmoncf/parmoncc**.

Also, there are parameters **perpass, peraver** defining the periods of data passing and averaging, respectively, as the number of minutes.

### 3.3 Function 'rnd128'

The double precision function **rnd128** is written using 64-bit integer arithmetic. The function has no arguments. After the initialization of the parallel RNG, **rnd128** starts returning base random numbers from a selected subsequence. Thus, on different processors, parallel streams of base random numbers are generated independently.

### 3.4 Command 'manaver'

The program **manaver** is used to average and to save in files the subtotal sample moments calculated on processors. It is launched after the termination of a job on a cluster. The application of **manaver** is useful in the case when by the moment of terminating the job, the sample moments stored in the files with results correspond to a smaller sample volume than to the one that was actually obtained on all the processors.

### 3.5 Command 'genparam'

If one wants to define different values of the parallel RNG multipliers $A(n_e)$, $A(n_p)$ and $A(n_r)$ in comparison with the default ones, he runs the program **genparam** from a command line in his working directory in the following way:

```
$ genparam ne np nr
```

where **ne, np** and **nr** are exponents of 2. As a result, a file **parmonc_genparam.dat** is created in the user's working directory. Hereupon, the PARMONC routines use the multipliers' values from this file instead of the default ones.

### 3.6 Description of Files with Results of Simulation

When the user launches a job on a cluster, a subdirectory **/parmonc_data** is automatically created by the PARMONC in his/her working directory. In the directory **/parmonc_data/results** one can find the results of computation stored in the files **func.dat**, **func_ci.dat** and **func_log.dat**:

- **func.dat** stores a matrix of the sample means,
- **func_ci.dat** encapsulates a matrix of the sample means together with matrices of absolute and relative errors and variances,
- **func_log.dat** stores information about the stochastic simulation: the total sample volume, the mean computer time per a realization, the upper bounds for absolute and relative errors, etc.

Also, in this directory, one can find a file **parmonc_exp.dat** containing information about each stochastic experiment that was started by the user.

## 4 Performance Test

The following example may be found in the full documentation to the PARMONC [4]. Also, it is available for the users of the the Siberian Supercomputer Center in the directory of the library [3].

We consider a 2-dimensional system of stochastic differential equations (SDEs) over a time interval $[0, 100]$:

$$d\bar{y}(t) = Cdt + Dd\bar{w}(t),$$

where

$$\bar{y}(0) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \bar{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix}, C = \begin{pmatrix} 1.0 \\ 1.0 \end{pmatrix}, D = \begin{pmatrix} 10^{-2} & 0 \\ 0 & 10^{-2} \end{pmatrix},$$

and $\bar{w}(t) = \begin{pmatrix} w_1(t) \\ w_2(t) \end{pmatrix}$ is a 2-dimensional Wiener process. Our objective is to evaluate expectations of its components $\mathrm{E}y_1(t)$, $\mathrm{E}y_2(t)$ at fixed points

$t_i = i \cdot 10^{-1},\ i = 1, \ldots, 1000$. We simulate trajectories of the SDE system using a generalized Euler method with a mesh size $h = 10^{-6}$:

$$\bar{y}^{(n+1)} = \bar{y}^{(n)} + hC + \sqrt{h} D \bar{\xi}^{(n)}, n = 0, 1, 2, \ldots, 10^8, \tag{9}$$

where

$$\bar{y}^{(0)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \bar{y}^{(n)} = \begin{pmatrix} y_1^{(n)} \\ y_2^{(n)} \end{pmatrix}, \bar{\xi}^{(n)} = \begin{pmatrix} \xi_1^{(n)} \\ \xi_2^{(n)} \end{pmatrix},$$

all $quantity\{\xi_i^{(n)}\}$ being independent in total and having a standard normal distribution. The simulation yields a matrix $[\zeta_{ij}]$:

$$\zeta_{ij} = y_j^{(n)}, n = i10^5, 1 \le i \le 1000, 1 \le j \le 2.$$

Thus, each entry of the matrix after averaging gives:

$$\bar{\zeta}_{ij} \approx \mathrm{E} y_j(t_i), t_i = i \cdot 10^{-1},\ i = 1, \ldots, 1000, j = 1, 2.$$

Below, as an example, the main user's program in C containing a call to **parmoncc** is provided.

```
int main()
{
   int nrow = 1000, ncol = 2, res = 1, seqnum = 2, perpass = 10,
       peraver = 20;
   long long int maxsv = pow(10,9);
   parmoncc ( difftraj, &nrow, &ncol, &maxsv, &res, &seqnum,
              &perpass, &peraver );
   return 0;
}
```

Here **difftraj** is the name of the user-supplied subroutine implementing the simulation of a realization of an approximate diffusion trajectory according to (9) and returning a realization of matrix $[\zeta_{ij}]$; **nrow** and **ncol** define dimensions of the matrix; **maxsv** is a maximal sample volume to simulate on processors; **res** is a resumption flag; **seqnum** defines the "experiments" subsequence number; **perpass** and **peraver** define the period of sending and receiving data, respectively (in minutes).

In this example **res = 1**. This means the case of resuming the previous simulation: the PARMONC automatically takes into account results of the previous simulation (from the corresponding files) and averages it by formulas (5). Also, **seqnum = 2**. This means that we use the "experiments" subsequence with number 2. Processors send subtotal data to the 0-th processor every 10 minutes. In turn, the 0-th processor receives data every 20 minutes. The argument **maxsv** is chosen to be sufficiently large in order to have an "endless" stochastic simulation that is limited only by the time framework of a job on a cluster (it is defined by the user when starting the job).

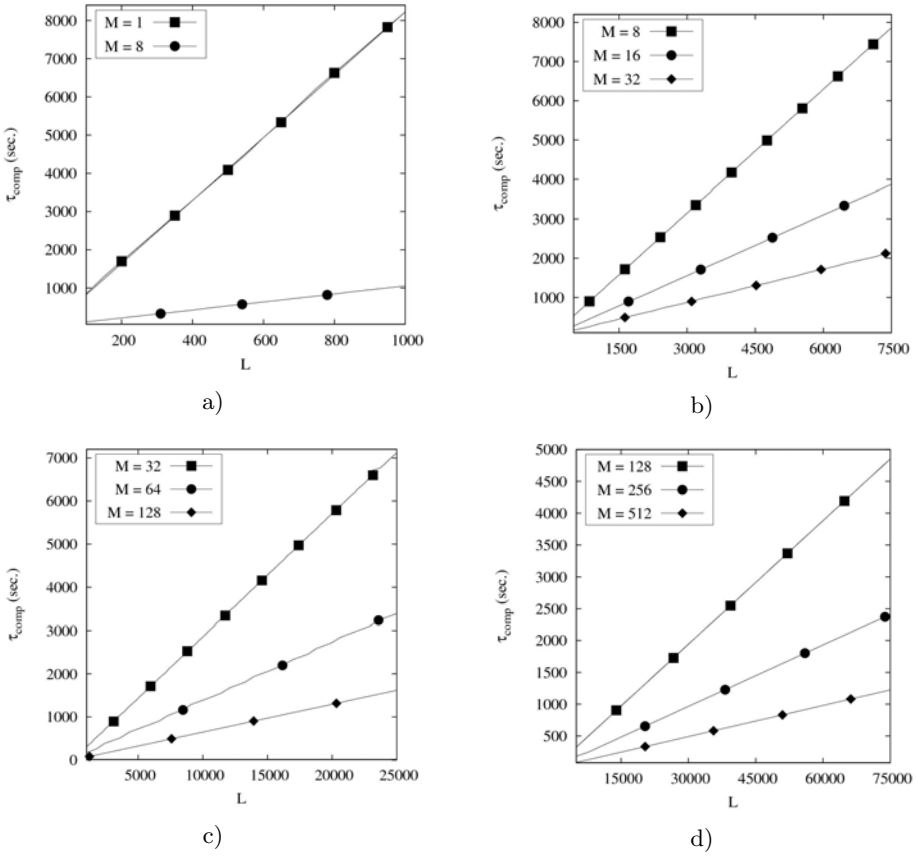



**Fig. 2.** Results of a PARMONC performance test: comparison of the computer time $\tau_{comp} = \tau_{comp}(L)$ for different numbers of processors: a) M = 1 and 8, b) M = 8, 16 and 32, c) M = 32, 64 and 128, d) M = 128, 256 and 512. In each graph X-axis corresponds to the total sample volume $L$, Y-axis corresponds to $\tau_{comp}$ measured in seconds.

In the subroutine **difftraj**, the parallel RNG is called in the following simple way:

```
a = rnd128();
```

This way of calling the RNG seems the most natural for specialists in the stochastic simulation.

The above-mentioned diffusion problem was computed on 1, 8, 16, 32, 64, 128, 256 and 512 processors to compare the speedup of parallelization. All the processors sent data to the 0-th processor after having simulated each realization. In turn, the 0-th processor received data after having simulated each realization. Such conditions are assumed to be strictest in terms of the parallel algorithm performance. A mean computer time $\tau_\zeta$ to simulate a single realization is

approximately 7.7 sec., the bulk of data which is periodically sent by every processor to the 0-th processor is approximately 120 Kbytes.

For different numbers of processors $M$ we compare the computer time it takes to simulate $L$ realizations in total $\tau_{comp} = \tau_{comp}(L)$ . A value of $\tau_{comp}$ is evaluated after the 0-th processor has received, averaged and saved data in files.

It is seen from the graphs in Fig. 2 that for all the values of $L$ the speedup of parallelization is in direct proportion to the number of processors despite "strict" conditions related to data exchange.

## 5 Conclusion

In conclusion, we define some directions for the future. First of all, it is desirable to adapt the PARMONC to modern powerful GPU computer clusters and, also, to hybrid computer clusters. Then, it seems promising to use the PARMONC as a basic software level for the future computer-aided simulation based on adequate probabilistic models to imitate real world phenomena from the "first principles".

**Acknowledgements.** The author would like to thank Prof. Victor Malyshkin for our helpful discussions on the subject of this paper; Nikolai Kuchin, Sergey Kireev and Maxim Gorodnichev for their valuable advice; Dr. Galiya Lotova for her patience and help in improving the reliability of the library. Also, the author would like to thank Prof. Boris Glinskiy, Head of the Laboratory "Siberian Supercomputer Center", for his kind attention to the present research.

## References

1. Martin, W.R.: Advances in Monte Carlo Methods for Global Reactor Analysis. In: Invited lecture at the M&C 2007 International Conference, Monterey, CA, USA, April 15-19 (2007)
2. Brown, F.B., Martin, W.R., Mosteller, R.D.: Monte Carlo - Advances and Challenges. In: Workshop at PHYSOR-2008, Interlaken, Switzerland, September 14-19, Report LA-UR-08-05891, Los Alamos National Laboratory (2008), http://www.physor2008.ch/documents/Workshop_I/PHYSOR08-WorkShopI.pdf
3. Page of PARMONC on the web site of Siberian Supercomputer Center, http://www2.sscc.ru/SORAN-INTEL/paper/2011/parmonc.htm
4. Link to a full documentation to PARMONC, http://www2.sscc.ru/SORAN-INTEL/paper/2011/parmonc.pdf
5. Marchenko, M.A., Mikhailov, G.A.: Distributed computing by the Monte Carlo method. Automation and Remote Control 68(5), 888–900 (2007)
6. Brent, R.: Fast and reliable random number generators for scientific computing. In: Dongarra, J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 1–10. Springer, Heidelberg (2006)
7. The Scalable Parallel Random Number Generators Library (SPRNG), http://sprng.fsu.edu/
8. Coddington, P.D., Newell, A.J.: JAPARA – A Java Parallel Random Number Generator Library for High-Performance Computing. In: 18th International Parallel and Distributed Processing Symposium (IPDPS 2004) - Workshop 5, vol. 6, p. 156a (2004)

9. Mendes, B., Pereira, A.: Parallel Monte Carlo Driver (PMCD) - a software package for Monte Carlo simulations in parallel. Comput. Phys. Comm. 151(1), 89–95 (2003)
10. Badal, A., Sempau, J.: A package of Linux scripts for the parallelization of Monte Carlo simulations. Comput. Phys. Comm. 175(6), 440–450 (2006)
11. Slawinska, M., Jadach, S.: MCdevelop - a universal framework for Stochastic Simulations. Comput. Phys. Comm. 182(3), 748–762 (2011)
12. Mikhailov, G.A., Voytishek, A.V.: Numerical stochastic simulation. Publishing Center "Akademia" (2006) (in Russian)
13. Rubinstein, R.Y., Kroese, D.P.: Simulation and the Monte Carlo Method, 2nd edn. John Wiley & Sons, New York (2007)
14. Dyadkin, I.G., Hamilton, K.G.: A study of 128-bit multipliers for congruential pseudorandom number generators. Comput. Phys. Comm. 125(1-3), 239–258 (2000)
15. Marchenko, M.A.: Parallel pseudorandom number generator for large-scale monte carlo simulations. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 276–282. Springer, Heidelberg (2007)
16. Page of 128-bit parallel congruential random number generator, Department of Stochastic Simulation in Physics of the Institute of Computational Mathematics and Mathematical Geophysics in Novosibirsk, Russia, http://osmf.sscc.ru/~mam/generator_en.htm

# On Performance Analysis of a Multithreaded Application Parallelized by Different Programming Models Using Intel VTune

Ami Marowka

Department of Computer Science
College of Exact Sciences
Bar-Ilan University, Ramat Gan, Israel 52900
amimar2@yahoo.com
http://www.cs.biu.ac.il

**Abstract.** Multi-core processors are ubiquitous. Extracting the desired performance from them requires efficient techniques for partitioning a single piece of work into multiple fine-grained units of work in order to process them simultaneously. Understanding the performance behavior of a parallel system requires a close familiarity with the underlying architecture and the hardware counters.

We present a performance analysis study of a multi-core system by a state-of-the-art parallel performance analyzer tool, the Intel VTune Performance Analyzer. We chose as a test-case a classic nested-loop application that exhibits unexpected performance gains using two different programming models on the same multi-core system. Our expectations were to be able to reason about the performance results by exploring the application behavior using the parallel analyzer tool. We found that it is very difficult to explain high-level performance measurements of multi-core systems by low-level hardware diagnosis.

**Keywords:** Multi-core, Performance Analysis, OpenMP, TBB.

## 1 Introduction

Multi-core processors can now be found in the heart of supercomputers, desktop computers and laptops [1]. Consequently, applications will increasingly need to be parallelized to fully exploit the multi-core processor throughput gains that are becoming available. Unfortunately, parallel code is more complex to write than that of serial code [2]. Parallel programming is no doubt much more tedious and error-prone than serial programming.

Chip makers and system builders have begun efforts to educate developers and provide them with better tools for multi-core programming. Currently, the responsibility for bridging the gap between hardware and software in order to write better parallel programs may ultimately lie with developers. Many programmers are not up to speed on the latest developments in hardware design.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 317–331, 2011.

They should study chip architectures to understand how their code can perform better. This is not a desirable situation. Parallel programming should be as simple and productive as sequential programming.

Task-based parallel programming aims to simplify the writing of parallel code. It is a state-of-the-art programming technique that has been adopted by the major software vendors as the primary approach for multi-core programming. Notable commercial products are Intel Threading Building Blocks [3,11]; Intel Cilk++ [4]; OpenMP [5,12] (since version 3.0); Microsoft Task Parallel Library [6] and Java Fork/Join Framework (JSR166) [7] alongside research projects such as TaskMan [8], Wool [9] and Nanos Mercurium [10].

The task-based programming approach offers an alternative to the traditional thread-based programming. Expressing parallelism with task-based programming is done by partitioning the application into fine-grained and independent executable entities called *tasks* rather than by managing explicit threads for processing coarse-grained units of work. The run-time system of a task-based environment handles all the synchronization and the scheduling assignments implicitly and thus frees the programmer to focus on the algorithmic design of the application. Writing code with tasks usually yields more simple, portable and efficient code as compared to thread-based code. It was found that creating and terminating a task is 18 times faster in comparison to creating and terminating a thread on Linux systems and up to 100 times faster on Windows systems [3].

The core engine of a task-based model is a task scheduler which uses a task stealing mechanism to balance a parallel workload across available processing cores in order to increase core utilization and overall system performance and scalability. After an initialization phase, the task scheduler decomposes the workload into tasks and stores them in distributed queues, usually one queue per core/thread. Then, the scheduler assigns a thread from a thread pool to one of the queues where tasks are waiting for processing. When a thread's own queue is empty, the thread steals work from another thread's queue. The donor thread is chosen randomly and the stolen task is the last one in the queue.

This paper presents a low-level performance study of a task-based application and its equivalent thread-based implementation on a multi-core processor. For this purpose, we chose a typical application that was parallelized by Intel TBB (a task-based model) and by OpenMP (a thread-based model). The test-case application, the Substring-Finder, was chosen because the performance results show that the TBB implementation outperforms the OpenMP implementation. These results are surprising because TBB is designed to be compiler independent, processor independent, and operating system independent. Therefore, it is expected that OpenMP will exhibit better performance than TBB. The opposite results brought us to explore the reasoning behind these unexpected performance results.

First, we measured the performance of both implementations. Then, a detailed low-level analysis was performed in order to study the behavior of the two implementations in the underlying processor architecture. The processor's memory sub-system behavior was studied by careful analysis of its raw

performance data which was collected from the processor's hardware counters by the Intel VTune Performance Analyzer tool [13]. To the best of our knowledge, this paper presents the first attempt to explore the reasoning about the performance differences of two different parallel programming paradigms by hardware events.

The rest of this paper is organized as follows. Section 2 is a brief introduction of OpenMP and TBB. Section 3 presents an in-depth performance analysis of a case-study application implemented by TBB and OpenMP. Section 4 is a brief discussion. Section 5 presents related works, and Section 6 concludes the paper.

## 2 Overview of TBB and OpenMP

OpenMP [5,12] and TBB [3,11] are two parallel programming paradigms suitable for multi-core processors. OpenMP and TBB have a lot in common but were designed for different parallel execution models. Both are shared-memory data-parallel programming models and are based on multi-threading programming to maximize utilization of multi-core processors. Furthermore, the TBB core execution model is based on Task programming approach while OpenMP is based on the Thread programming approach, supporting task-based parallelism since its version 3.0 (2008).

OpenMP does not free the programmer from most of the tedious issues of parallel programming. The programmer has much to understand, including: the relationship between the logical threads and the underlying physical processors and cores; how threads communicate and synchronize; how to measure performance in a parallel environment; and the sources of load unbalancing. The programmer must check for dependencies, deadlocks, conflicts, race conditions, and other issues related to parallel programming. In contrast, the Intel Threading Building Blocks hides some of the issues mentioned above from the programmer and automates the data decomposition and tasks scheduling in an efficient manner. This section is a brief overview of the TBB and OpenMP parallel programming models.

### 2.1 Threading Building Blocks (TBB)

The Intel Threading Building Blocks (TBB) is a C++ template library that supports data parallel programming for developing parallel applications running on top of multi-core processors. TBB designers are committed to make TBB compiler independent, processor independent, and operating system independent. The library consists of building blocks (data structures and algorithms) that free a programmer from some of the complications arising from the use of native threading mechanisms such as threads creation, synchronization, and termination.

TBB allows the programmer to design an application in terms of task objects. Parallelism is expressed explicitly by using TBB constructs while nested parallelism is allowed. However, the programmer is responsible for building independent and thread-safe tasks. TBB's task scheduler maps user-defined logical

tasks onto physical threads in a one-to-one manner, that is, one software thread per hardware thread.

The task scheduler uses an efficient and dynamic load balancing mechanism based on task-stealing for improving performance throughput. For this purpose, each thread maintains a queue of tasks. A thread performs depth-first execution while using its local queue as a stack and thus achieves low footprint space and good data locality. If a thread runs out of work it is allowed to steal a task, usually a big chunk, from the rear side of a busy thread queue.

TBB library consists of algorithms (parallel for, parallel reduce, parallel scan, parallel while, pipeline, and parallel sort). The design of the algorithms is based on C++ generic programming and recursively divisible ranges implemented on top of an efficient work-stealing scheduler. The library was designed for simplicity while parallelism is mapped to the underlying machine resources without the intervention of the programmer.

In addition, TBB library provides concurrent containers (concurrent queue, concurrent vector, and concurrent hash map). The containers are thread-safe and provide fine-grained locking for efficiency. The library also contains concurrent memory allocation, various mutual exclusion mechanisms, and atomic operations.

### 2.2 OpenMP Programming Model

OpenMP is a tool for writing multi-threaded applications in a shared memory environment. It consists of a set of compiler directives and library routines. The compiler generates a multi-threaded code based on the specified directives.

OpenMP helps developers create multithreaded applications more easily while retaining the look and feel of serial programming. OpenMP simplifies the complex task of code parallelization, allowing even beginners to move gradually from serial programming styles to parallel programming. OpenMP extends serial code by using compiler directives. A programmer familiar with a language (such as C/C++) needs to learn only a small set of directives. Adding them does not change the logical behavior of the serial code; it tells the compiler only which piece of code to parallelize and how to do it; the compiler handles the entire multithreaded task.

An OpenMP program begins with a single thread of execution called the master thread. The master thread spawns teams of threads in response to OpenMP directives, which perform work in parallel. Parallelism is thus added incrementally: the serial program evolves into a parallel one. OpenMP directives are inserted at key locations in the source code. These directives take the form of comments in FORTRAN and pragmas in C and C++. The compiler interprets the directives and creates the necessary code to parallelize the indicated tasks/regions. The parallel region is the basic construct that creates a team of threads and initiates parallel execution.

OpenMP provides several constructs for sharing work among threads in a team. These are: Parallel for/DO, Parallel Sections, Workshare, and Single directive. These constructs are placed inside an existing parallel region. The result

is to distribute execution of associated statements among the existing threads. In addition, OpenMP provides a number of constructs for thread synchronization and coordination, among them critical, atomic, barrier, and master. These are sufficient for many needs, but OpenMP also provides a set of runtime thread-locking functions that can be used for fine control.

## 3 A Low-Level Analysis

In this section we describe the case study application and the elapsed time running results obtained from running it on top of a multi-core processor. Next, we explore the low-level hardware measurements to better understand the high-level performance results.

### 3.1 Case Study: Substring Finder

We chose the Substring-Finder application to be our case study which is one of the examples included in the TBB software suite, and it represents many classic nested-loop applications where a one-dimensional array is processed intensively. Such parallel algorithms are categorized as data-parallel applications. The Substring-Finder application scans a string, and for each location along the string it looks for the largest matching substring elsewhere in the string. For each match, the application stores the location and the length of the substring.

*Listing 1 - Substring-Finder - Serial Version.*

```
void SerialSubStringFinder ( const string &str, size_t *max_array,
                             size_t *pos_array) {
 for ( size_t i = 0; i < str.size(); ++i ) {
   size_t max_size = 0, max_pos = 0;
   for (size_t j = 0; j < str.size(); ++j)
    if (j != i) {
     size_t limit = str.size()-( i > j ? i : j );
     for (size_t k = 0; k < limit; ++k) {
      if (str[i + k] != str[j + k]) break;
      if (k > max_size) {
       max_size = k;
       max_pos = j;
      }
     }
    }
   max_array[i] = max_size;
   pos_array[i] = max_pos;
  }
}
```

Listing 1 shows the serial version of the application. Listing 2 is the TBB parallel version. The listings show only the core portions of the applications.

The TBB version uses the *parallel_for* template for invocation of the operator of the class SubstringFinder where the work on the string is partitioned with a *blocked_range* with a *grain-size* of 100. Reinders reports that the task scheduler splits the string into 254 tasks on a dual-core processor[3]. Listing 3 is the OpenMP parallel version of the application. The OpenMP and the serial versions are nearly identical. Only one OpenMP compiler directive was needed to annotate the serial code in this case. The result is a very simple and compact way to parallelize an existing code as compared to the TBB version where the code has to be re-written.

*Listing 2 - Substring-Finder - TBB Version.*

```
class SubStringFinder {
 const string str;
 size_t *max_array;
 size_t *pos_array;
 public:
 void operator() ( const blocked_range<size_t>& r ) const {
  for ( size_t i = r.begin(); i != r.end(); ++i ) {
   size_t max_size = 0, max_pos = 0;
   for (size_t j = 0; j < str.size(); ++j)
    if (j != i) {
     size_t limit = str.size()-( i > j ? i : j );
     for (size_t k = 0; k < limit; ++k) {
      if (str[i + k] != str[j + k]) break;
      if (k > max_size) {
       max_size = k;
       max_pos = j;
       }
      }
     }
   max_array[i] = max_size;
   pos_array[i] = max_pos;
  }
 }
 SubStringFinder(string &s, size_t *m, size_t *p) :
  str(s), max_array(m), pos_array(p) { }
};

int main(int argc, char *argv[]) {
.
parallel_for(blocked_range<size_t>(0, to_scan.size(), 100),
       SubStringFinder( to_scan, max, pos ) );
...}
```

*Listing 3 - Substring-Finder - OpenMP Version.*

```
void OmpSubStringFinder ( const string &str, size_t *max_array,
                          size_t *pos_array) {
#pragma omp parallel for num_threads(2)
for ( int i = 0; i < str.size(); ++i ) {
 int max_size = 0, max_pos = 0;
 for (int j = 0; j < str.size(); ++j)
  if (j != i) {
   int limit = str.size()-( i > j ? i : j );
    for (int k = 0; k < limit; ++k) {
     if (str[i + k] != str[j + k]) break;
       if (k > max_size) {
         max_size = k;
         max_pos = j;
     }
    }
   }
max_array[i] = max_size;
pos_array[i] = max_pos;
 }
}
```

### 3.2 Experimental Results

We tested the performance of the three versions of the Substring-Finder application on a dual-core machine. The machine was equipped with an Intel Core 2 Duo processor (T8100) with a clock speed of 2.1GHz; 2x32KB L1 data cache memory and 2x32KB L1 instructions cache memory; 1x3MB L2 unified shared cache memory and 1GB DDR2 main memory. For software we used OpenMP 3.0 and TBB 3.0 under Intel C++ compiler XE version 12.0 on top of an XP operating system. We used Intel VTune Performance Analyzer version 9.1 for Windows and the new Intel VTune Amplifier XE for Windows to collect data for the low-level analysis. The compilers optimization option was set to -O2 . The elapsed times were measured for two Fibonacci strings, a short string (17,771 bytes), and a long string (317,811 bytes).

The results reported here are averages of ten runs each for statistical stability. Table 1 shows the running results of the serial, OpenMP and TBB Substring-Finder versions for short and long string sizes. The following findings can be observed from these measurements:

- TBB achieves better speedup as compared to OpenMP for both string sizes while exhibiting an improvement of up to 27%.
- TBB achieves super-linear speedup ( $> 2$) which is an indication for an efficient use of the cache-memory hierarchy.

**Table 1.** Measurement times (in seconds) and computed speedups of Substring-Finder

| String Size | Serial | TBB (1 thread) | TBB (2 thread) | OMP (1 thread) | OMP (2 thread) | TBB Speedup | OMP Speedup |
|---|---|---|---|---|---|---|---|
| 17,711 | 23.90 | 20.78 | 11.53 | 26.03 | 14.50 | 2.07 | 1.64 |
| 317,811 | 10,277 | 8,874 | 4,977 | 11,246 | 6321 | 2.06 | 1.62 |

– TBB performance with one thread achieves better performance as compared to the serial version (up to 13% improvement). On the other hand, OpenMP performance with one thread achieves the worst performance as compared to the serial version (up to 10% reduction). These observations show that OpenMP adds substantial parallelism overhead while TBB's task-scheduler partitions the workload into small tasks and schedule them more efficiently among the processing cores.

In our attempt to discover the reasons behind the low performance of OpenMP compared to TBB we performed the following actions:

**Thread Affinity.** During the performance exploration, it was observed that TBB binds threads to physical processing cores while OpenMP does not enforce thread affinity. This observation is visualized very nicely by the Time-based sampling capability viewer of Intel Vtune Performance Analyzer version 9.1 (Unfortunately, with the new Intel VTune Amplifier XE for Windows it can be observed indirectly since thread performance data can't be associated to specific core). Therefore, we enforced thread-affinity on the OpenMP implementation by using the operating system *SetTreadAffinityMask()* function. Intel VTune showed clearly that the threads were bound to the appropriate cores. Unfortunately, we did not observe any improvement in performance.

**Loop Scheduling.** We tried to improve the performance of the OpenMP implementation by enforcing different scheduling policies. The Static, Dynamic and the Guided OpenMP loop scheduling were tested with different chunk sizes. Unfortunately, we cannot report on significant improvement in performance.

**Compiler Optimization Options.** We tested the OpenMP implementation with different compiler optimization options: Maximize Speed (/O2), Maximize Speed plus High Level Optimization (/O3) and Full Optimization (/Ox). Unfortunately, the performance measurements did not show any improvement.

**Str.size().** We suspected that the calls to *str.size()* that appear in the for-loops increase the overhead and thus decrease the performance. However, we tested two other alternatives and found that the compiler's optimizer creates more efficient code when using the *str.size()* rather than define a constant instead the function call or using a variable that is assigned once outside the parallel region.

### 3.3 Hardware Events

In order to understand why TBB outperforms OpenMP in the case of the Substring-Finder, we further explored the low-level performance characteristics

of the two implementations. For this purpose we used the new Intel VTune Amplifier Performance Analyzer. VTune samples the processor counters and collects hardware events. These events are recorded, analyzed, and displayed to the user for further exploration. We chose to follow the recommendation given by Ulrich Drepper [14,15], to inspect the memory subsystem effects by using a few event ratios rather than absolute values because it makes the interpretation easier. The following are brief descriptions of the event ratios we used in our low-level measurements:

- CPI – This ratio represents the number of clocks per instructions retired. Lower CPI indicates the better efficiency with which instructions are executed within the processor.
- L1D_REPL % – This ratio represents the number of L1 data cache misses. A high rate indicates that pre-fetching is not effective.
- L1I_MISSES % – This ratio represents the number of L1 instruction misses. A high rate indicates either unpredictable execution flow or that the code size is too large.
- DTLB_MISSES.ANY % – This ratio represents the number of Data Table Lookaside Buffer (DTLB) misses. A high rate indicates that the code accesses too many data pages within a short time.
- ITLB_MISS_RETIRED % – This ratio represents the number of retired instructions that missed the ITLB. A high rate indicates that the executed code is spread over too many pages.
- L2_LINES_IN % – This ratio represents the number of L2 cache misses. A high rate indicates that the L2 cache cannot hold the data.
- L2_IFETCH % – This ratio represents the number of L2 instruction misses. Any rate above zero may have a noticeable impact on application performance.
- PAGE_WALKS.CYCLES % – This ratio represents the number of cycles spent waiting for page-walks. A high rate indicates that many cycles are spent on TLB misses.
- STORE_BLOCK.ORDER % – This ratio represents the number of cycles for which stores are waiting for a preceding stored cache line to be observed by other cores. A high rate indicates that too many store operations miss the L2 cache and therefore block committing data of later stores to the memory.
- CYCLES_L1I_MEM_STALLED % – This ratio represents the number of cycles for which an instruction fetches stalls. A high rate indicates poor code locality.
- RESOURCE_STALLS.RS_FULL % – This ratio represents the number of cycles in which the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high rate indicates poor efficiency of the parallelism provided by the multiple execution units.

Table 2 shows the event ratios results of OpenMP and TBB Substring-Finder implementations for short and long string sizes. The following findings can be observed from these measurements.

In the case of the short string, where the entire string is held in the L1 cache, TBB exhibits better utilization of the cache-memory subsystem. TBB presents improvement in 3 of 11 ratios as compared to OpenMP. The most noticeable improvements are less L1 data and instruction misses (53% and 33% improvements respectively), less TLB data and instruction misses (58% and 53% improvements respectively) and 15% improvement of the CPI. These findings explain the elapsed times measurements of TBB as compared to OpenMP and the way TBB achieves better speedup. TBB succeeds in adapting its data chunks more efficiently into the cache-memory subsystem as compared to OpenMP, needing fewer clocks per instruction for executing its code. On the other hand, OpenMP code tends to stall for longer period of time, which indicates that it spreads over many pages.

In the case of the long string, the results are changed upside down. OpenMP presents improvement compared to TBB in 3 of 11 ratios. The most noticeable improvements are less L2 and TLB data misses (20% and 13% improvements respectively) and less time delays due to page-walks, store stalls and L1 instruction stalls (18%, 25% and 19% improvements respectively).

Now, the unavoidable question must be raised. How does TBB achieve better speedup in the case of the long string when the event ratios results indicate that the opposite should have happened?

First, it can be observed that there is a dramatic increase in all ratios of the long string case as compared to the short string case. This happens due to the use of inclusive caches and because the long string is spread over the L1 and L2 caches. Second, TBB's task-scheduler creates more tasks for a longer workload, and the result is more L1, L2 and TLB data misses that lead to more instruction stalls. However, these misses and stalls cannot explain why TBB still achieves super-linear speedup. The CPI rations for the short and the long cases show that TBB needs approximately 17% less clocks per instruction as compared to OpenMP. TBB produces more efficient code that compensates for the addition of misses and stalls in its many tasks. However, the CPI only reflects the results but cannot explain them.

## 4 Discussion

Intel VTune Performance Analyzer is considered the de-facto performance analyzer tool for parallel applications that are created using Intel and Microsoft Windows developing environments. VTune's interface is very user-friendly, colorful and includes readable graphic performance views. It is possible to compare very easily the hardware measurements of two running results of two versions of the same application on the same platform. However, it is impossible to make low-level comparison of two different parallel programming paradigms implementations of the same application on the same platform.

Intel VTune is actually a tread-base tool and it is not a task-based tool. It is very convenient to use it for analyzing the performance of individual thread but there is no way to view tasks or to display a time-based view of a specific task.

**Table 2.** Event ratios results for Substring Finder application

| Event | | | | |
|---|---|---|---|---|
| String Size | 17,771 | | 317,811 | |
| | TBB | OMP | TBB | OMP |
| CPI | 0.54 | 0.64 | 0.53 | 0.63 |
| L1D_REPL % | 10.96 | 23.81 | 90.91 | 87.85 |
| L1I_MISSES % | 3.14 | 4.71 | 4.62 | 4.80 |
| DTLB_MISSES.ANY % | 25.53 | 61.35 | 50.94 | 44.1 |
| ITLB_MISS_RETIRED % | 3.95 | 8.51 | 5.30 | 6.56 |
| L2_LINES_IN % | 8.84 | 7.23 | 10.33 | 8.29 |
| L2_IFETCH % | 4.17 | 7.69 | 6.57 | 6.85 |
| PAGE_WALKS.CYCLES % | 11.34 | 11.76 | 19.14 | 12.59 |
| STORE_BLOCK.ORDER % | 23.53 | 23.44 | 16.34 | 12.30 |
| CYCLES_L1I_MEM_STALLED % | 17.12 | 22.41 | 19.83 | 9.18 |
| RESOURCE_STALLS.RS_FULL % | 98.75 | 97.64 | 98.78 | 97.64 |

For using VTune in an appropriate manner, the programmer has to know where and what to look for in order to find performance inhibitors. Moreover, the programmer has to understand performance issues such as load-balancing, false-sharing and different types of parallelization overheads.

## 5 Related Works

There are many research papers that used Intel VTune Performance Analyzer as a tool for exploring different aspects of performance behavior. This section presents a research that studied the basic parallelism management of TBB [16]; a study that compares the scalability of Windows Threads versus TBB [17]; a work that studied two different optimization strategies aim to improve the performance of the work-stealing task scheduler of TBB [18]; studies of OpenMP and TBB applications from different disciplines and from various angles (But none of them used hardware events to reason about the performance measurements) [19,20]; a research project of a new library that supports task-based parallel model [8]; a work that presents one of the weaknesses of current performance analyzing tools to explore the sources of performance bottlenecks and suggest a way to resolve the problem [21] and performance comparison studies of compilers by using Intels VTune Performance Analyzer [22,23].

Contreras and Martonosi studied the basic parallelism management costs of the TBB runtime Library [16]. On the hardware side, their tested platform was a quad-core machine for measuring performance on 1-4 real cores and simulations for study the overheads on 4-32 virtual cores. On the software side, they used four micro-applications of the PARSEC benchmark suite (*Fluidanimate, Swaptions, Blackscholes, and Streamcluster*) that were ported to TBB and four kernel-benchmarks (*Bitcounter, Matmult, LU and Treeadd*). In measuring the basic operations of the TBB runtime library, they focused on five common operations: *Spawn, Get_task, Steal, Acquire_queue and Wait_for_all*.

Analysis of the benchmarking results discovers the following findings: synchronization overheads within TBB have a significant impact on parallelism performance; the runtime library contributes up to 47% of the total per-core execution time on a 32-core system (due to synchronization overhead within the TBB scheduler) and hinders performance scalability; the random task stealing mechanism becomes less effective as application heterogeneity and core counts increase; and a queue occupancy-based stealing policy can improve performance of task stealing by up to 17%.

Wang and Xu [17] studied the scalability of the multiple-pattern matching algorithm known as Aho-Corasick-Boyer-Moore Algorithm on Intel Core 2 Duo processor 6300, 1.86 GHz with 1GB main memory with Windows XP for different input sizes. This work compares the performance of the Windows Threading API against Intel Threading Building Blocks. The authors found that the average scalability achieved by TBB is 1.655 compare to 1.549 of Win32. The authors explain that TBB achieves better performance because TBB specifies tasks instead of threads. A task can be assigned to a thread dynamically; Intel TBB selects the best thread for a task by using the task scheduler. If one thread runs faster, it is assigned to perform more tasks. However, with the Win32 Threading Library, a thread is assigned to a fixed task, and it can not be reassigned to other tasks even though it is idle.

Robison et al [18] studied two different optimization strategies aim to improve the performance of the work-stealing task scheduler of TBB. The first optimization automatically tunes the grain size based on inspection of the stealing behavior. The second optimization improves cache affinity by biased stealing. For testing the impact of the grain size on the performance the authors used $Pi$, a simple numeric integration benchmark that computes $\pi$. The results show that OpenMP *static* achieves the best speedup compared to all OpenMP and TBB scheduling strategies while TBB *affinity_partitione* achieves the best speedup compared to other TBB strategies. The cache affinity benchmark used two programs: Sesmic, a program that simulates 2D wave propagation and Cholesky, a program that decomposes a dense symmetric square array A into lower triangular matrix L such that $A = LxL^T$. Only OpenMP *static* scheduling and TBB *affinity_partitioner* showed good results for more than two threads. OpenMP's static scheduling achieved the best result.

Kegel et al. [19] used different OpenMP and TBB implementations of a block-iterative algorithm for 3D image reconstruction (ListMode Ordered Subset Expectation Maximization) for comparison between OpenMP and TBB with respect to programming effort, programming style, and performance gain. The authors studied five implementations (two OpenMP and three TBB) that differ in the locking mechanisms used for synchronization. One of the OpenMP implementations used the atomic mutual exclusion mechanism while the second one used the critical mechanism. The three TBB implementations used mutex, queuing_mutex, and spin_mutex mechanisms respectively.

The authors found that OpenMP offers a simpler way to parallelize existing sequential code as compared to TBB, which demands the redesign and rewriting of

the sequential code and thus is more appropriate for building parallel programs from scratch. Moreover, although the two programming paradigms support high-abstraction APIs, OpenMP outperforms TBB while achieving better scalability due to more efficient locking mechanisms.

Podobas et al. [20] studied the performance results of five applications (FFT, NQueens, Multisort SparseLU, and Strassen) with six implementations of task-parallel programming models (four implementations of OpenMP, Cilk++ and Wool). The OpenMP implementations used four different compilers (GCC, Intel, Sun, and Mercurium). On the low-level, they studied the costs of task-creation and task-stealing.

The authors found that Wool and Cilk++ achieve low overheads in task-spawning and task-stealing as compared to OpenMP. Among the four implementation of OpenMP, the Intel compiler exhibited the lowest overheads. They also measured relative high overheads with respect to the parallel constructs of the tested applications. Even so, for coarse-grained tasks all the applications achieved high speedups. The authors say that most of their findings are still not clear and need further investigation.

Hower and Jackson [8] offer a C++ library called TaskMan that realizes the task-based programming paradigm. The library makes use of futures to introduce a call/return API that is usually found in imperative languages. The primary designing goals of TaskMan were simplicity and programmability. A performance study shows that TaskMan does not achieve the performance level of TBB and Cilk++ but has potential to close the gap while delivering a lightweight and intuitive API.

Tallent and Mellor-Crummey introduce one of the weaknesses of current profiling tools to explore the sources of performance bottlenecks and suggest a way to resolve the problem [21]. The problem the authors arise is related to contemporary task-based programming that relies on work-stealing techniques. They offer a low-overhead profiling scheme that quantifies parallel idleness (when threads wait for work) and overhead (when threads work on non-user code) and then marks the places in the users application that need more or less parallelism.

To achieve low-overhead scheme, the authors' solution is based on creating call path profiling by sampling rather than by instrumentation. However, sampling cannot produces reliable results due to unpredictable call path structure caused by the work-stealing mechanism. Therefore, the authors defined an idleness metric that takes in account the number of active working threads that do not supply enough tasks to keep all workers busy. Moreover, for measuring parallel overhead accurately there is a need to distinguish overhead instructions from useful-work instructions. For that purpose, the authors propose that the compiler will tag instructions that are associated with parallelization overhead. Then, a postmortem analysis tool will use these tags to identify instructions associated with overhead.

Tanjore and Milenkovic present a performance comparison study of Intel C++ compiler versus Microsoft VC++ compiler by examining the execution characteristics of the SPEC CPU 2000 benchmarks [22]. The performance metrics were

collected using Intels VTune Performance Analyzer. Although the elapsed time measurements show that the Intel C++ compiler performed better than VC++ for all the tested applications, detailed analyzing of execution characteristics such as instructions retired, loads retired, stores retired, branches retired, I-cache misses and mis-predicted branches do not reflect the running wall-clock times and thus cannot explain why Intel C++ compiler achieves better performance.

Prakash and Peng [23] studied the performance characterization of SPEC CPU2006 suite on Intel Core 2 Duo processor compiled by two compilers: Intel C++ and Microsoft VC++ compilers. The aim of this work is to compare between SPEC CPU2006 benchmarks and SPEC CPU2000 benchmarks and to make detailed performance comparison of the two compilers. The performance study was conducted by using Intel VTune Performance Analyzer. The authors found that CPU2006 benchmarks have larger input dataset and longer execution time than those of CPU2000. Moreover, it was observed that only 2 of 15 benchmarks (hmmer and h264ref) show better performance while compiled with Intel C++. The authors present very detailed measurements of the hardware counters and the hardware events. However, these interesting measurements do not explain the performance differences.

## 6 Conclusions

Task-based programming is becoming the state-of-the-art method of choice for extracting the desired performance from multi-core chips. It expresses a program in terms of light-weight logical tasks rather than heavy-weight threads.

In this paper we presented a case study where the performance of a task-programming model (TBB) was compared and analyzed against a thread-based programming model (OpenMP). The high-level results show that TBB outperforms OpenMP while the low-level analysis explains that TBB succeeds that due to producing efficient code.

Understanding the performance characteristics of an application by using profiling and analyzing tools such as Intel VTune is not an easy task. It requires extensive knowledge about the processors themselves to interpret data that usually contains absolute values collected from the hardware counters of the processors. The user should know the meaning of the various events and how they related to each other.

## References

1. Marowka, A.: Parallel Computing on Any Desktop. Communication of ACM 50(9), 74–78 (2007)
2. Marowka, A.: Pitfalls and Issues of Manycore Programming. Advances in Computers 79, 71–117 (2010)
3. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc, Sebastopol (2007)
4. Leiserson, C.E.: The Cilk++ concurrency platform. In: 46th Design Automation Conference, San Francisco, CA (2009)

5. OpenMP API, Version 3.0 (2008), http://www.openmp.org
6. Leijen, D., Hall, J.: Optimize managed code for multi-core machines (2007), http://msdn.microsoft.com/msdnmag/issues/07/10/futures/default.aspx
7. Java Fork/Join Framework (JSR166), http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/
8. Hower, D., Jackson, S.: TaskMan: Simple Task-Parallel Programming, http://pages.cs.wisc.edu/~david/courses/cs758/Fall2009/includes/Projects/JacksonHower-slides.pdf
9. Faxan, K.-F.: Wool user's guide, Technical report, Swedish Institute of Computer Science (2009)
10. Balart, J., Duran, A., Gonzalez, M., Martorell, X., Ayguada, E., Labarta, J.: Nanos mercurium: a research compiler for openmp. In: The Proceedings of the European Workshop on OpenMP (2004)
11. TBB Web Site, http://www.threadingbuildingblocks.org/
12. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP, Portable Shared Memory Parallel Programming. MIT Press, Cambridge (2007)
13. Intel VTune Performance Analyzer, http://software.intel.com/en-us/intel-vtune/
14. Drepper, U.: What Every Programmer Should Know About Memory (2007), http://people.redhat.com/drepper/cpumemory.pdf
15. Drepper, U.: Understanding Application Memory Performance. In: RED-HAT (2008)
16. Contreras, G., Martonosi, M.: Characterizing and Improving the Performance of Intel Threading Building Blocks. In: IEEE Proceeding of International Symposium on Workload Characterization, pp. 57–66 (2008)
17. Robison, A., Voss, M., Kukanov, A.: Optimization via Reflection on Work Stealing in TBB. In: Proceeding of IEEE International Symposium on Parallel and Distributed Processing, IPDPS, pp. 1–8 (2008)
18. Wang, L., Xu, X.: Parallel Software Development with Intel Threading Analysis Tools. Intel Technology Journal 11(04), 287–297 (2007)
19. Kegel, P., Schellmann, M., Gorlatch, S.: Using openMP vs. Threading building blocks for medical imaging on multi-cores. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 654–665. Springer, Heidelberg (2009)
20. Podobas, A., Brorsson, M., Faxan, K.: A Comparison of some recent Task-based Parallel Programming Models. In: The proceeding of the Third Workshop on Programmability Issues for Multi-Core Computers, Pisa, Italy, January 24 (2010)
21. Nathan, R.T., Mellor-Crummey, J.M.: Identifying Performance Bottlenecks in Work-Stealing Computation. IEEE Computer, 44–50 (December 2009)
22. Gurumani, S.T., Milenkovic, A.: Execution Characteristics of SPEC CPU2000 Benchmarks: Intel C++ vs. Microsoft VC++. In: ACM SE 2004, Huntsville, Alabama, USA, April 2-3, pp. 261–266 (2004)
23. Prakash, T.K., Peng, L.: Performance characterization of SPEC CPU2006 on Intel core 2 duo processor. In: ISAST 2008, vol. 2(1), pp. 36–41 (2008)

# Using Multidimensional Solvers for Optimal Data Partitioning on Dedicated Heterogeneous HPC Platforms

Vladimir Rychkov, David Clarke, and Alexey Lastovetsky

School of Computer Science and Informatics, University College Dublin,
Belfield, Dublin 4, Ireland
{Vladimir.Rychkov,Alexey.Lastovetsky}@ucd.ie,
David.Clarke.1@ucdconnect.ie

**Abstract.** High performance of data-parallel applications on heterogeneous platforms can be achieved by partitioning the data in proportion to the speeds of processors. It has been proven that the speed functions built from a history of time measurements better reflect different aspects of heterogeneity of processors. However, existing data partitioning algorithms based on functional performance models impose some restrictions on the shape of speed functions, which are not always satisfied if we try to approximate the real-life measurements accurately enough. This paper presents a new data partitioning algorithm that applies multidimensional solvers to numerical solution of the system of non-linear equations formalizing the problem of optimal data partitioning. This algorithm relaxes the restrictions on the shape of speed functions and uses the Akima splines for more accurate and realistic approximation of the real-life speed functions. The better accuracy of the approximation in its turn results in a more optimal distribution of the computational load between the heterogeneous processors.

**Keywords:** dedicated heterogeneous HPC platforms; data partitioning; functional performance models of processors; Akima spline interpolation; multidimensional root-finding.

## 1 Introduction

In this paper, we study partitioning of computational load in data-intensive parallel scientific routines, such as linear algebra, mesh-based solvers, image processing. In these routines, typically, computational workload is directly proportional to the size of data. High performance of these routines on dedicated heterogeneous HPC platforms can be achieved when all processors complete their work within the same time. This is achieved by partitioning the computational workload and, hence, data unevenly across all processors, with respect to the processor speed and memory hierarchy.

Conventional algorithms for distribution of computations between heterogeneous processors are based on a performance model which represents the speed of a

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 332–346, 2011.

processor by a constant positive number, and computations are distributed between the processors in proportion to this speed of the processor. The constant characterizing the performance of the processor is typically its relative speed demonstrated during the execution of a serial benchmark code solving locally the core computational task of some given size.

The fundamental assumption of the conventional algorithms based on the constant performance models (CPMs) is that the absolute speed of the processors does not depend on the size of the computational task. This assumption proved to be accurate enough if:

- The processors, between which we distribute computations, are all general-purpose ones of the traditional architecture,
- The same code is used for local computations on all processors, and
- The partitioning of the problem results in a set of computational tasks that are small enough to fit into the main memory of the assigned processors and large enough not to fit into the cache memory.

These conditions are typically satisfied when medium-sized scientific problems are solved on a heterogeneous network of workstations. Actually, heterogeneous networks of workstations were the target platform for the conventional heterogeneous parallel algorithms. However, the assumption that the absolute speed of the processor is independent of the size of the computational task becomes much less accurate in the following situations:

- The partitioning of the problem results in some tasks either not fitting into the main memory of the assigned processor and hence causing paging or fully fitting into faster levels of its memory hierarchy (Fig. 1).
- Some processing units involved in computations are not traditional general-purpose processors (say, accelerators such as GPUs or specialized cores). In this case, the relative speed of a traditional processor and a non-traditional one may differ for two different sizes of the same computational task even if both sizes fully fit into the main memory.
- Different processors use different codes to solve the same computational problem locally.

The above situations become more and more common in modern and especially perspective high-performance heterogeneous platforms. As a result, applicability of the traditional CPM-based distribution algorithms becomes more restricted. Indeed, if we consider two really heterogeneous processing units $P_i$ and $P_j$, then the more different they are, the smaller will be the range $R_{ij}$ of sizes of the computational task where their relative speed can be accurately approximated by a constant. In the case of several different heterogeneous processing units, the range of sizes where CPM-based algorithms can be applied will be given by the intersection of these pair-wise ranges, $\bigcap_{i,j=1}^{p} R_{ij}$. Therefore, if a high-performance computing platform includes a

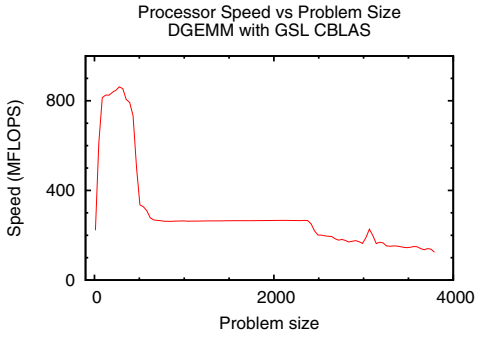



**Fig. 1.** Processor speed observed for matrix multiplication routine in different ranges of problem size

relatively large number of significantly heterogeneous processing units, the area of applicability of CPM-based algorithms may become quite small or even empty. For such platforms, new algorithms are needed that would be able to optimally distribute computations between processing units for the full range of problem sizes.

The functional performance model (FPM) of heterogeneous processors proposed and analyzed in [1] has proven to be more realistic than the constant performance models because it integrates many important features of heterogeneous processors such as the architectural and platform heterogeneity, the heterogeneity of memory structure, the effects of paging and so on. The algorithms employing it therefore distribute the computations across the heterogeneous processing units more accurately than the algorithms employing the constant performance models. Under this model, the speed of each processor is represented by a continuous function of the size of the problem. This model is application centric because, generally speaking, different applications will characterize the speed of the processor by different functions.

The cost of experimentally building the full functional performance model of a processor, i.e., the model for the full range of problem sizes, is very high. This is due to several reasons. To start with, the accuracy of the model depends on the number of experimental points used to build it. The larger the number, the more accurate the model is. However, there is a cost associated with obtaining an experimental data point, which requires execution of a computational kernel for a specified problem size. This cost is especially high for problem sizes in the region of paging.

The high model-construction cost limits the applicability of parallel algorithms based on full FPMs to situations where the construction of the full FPMs of heterogeneous processors and their use in the application can be separated. For example, if we develop an application for dedicated stable heterogeneous platforms, with the intention of executing the application on the same platform multiple times, we can build the full FPMs for each processor of the platform once and then use these models multiple times during the repeated execution of the application. In this case, the time of construction of the FPMs can become very small compared to the accumulated performance gains during the multiple executions of the optimized application. However, this approach does not apply to applications for which each run is considered unique. This is the case for applications that are intended to be executed

in dynamic environments or any other environments where the available processors and their performance characteristics can change.

In contrast to the CPM-based data partitioning algorithms, the FPM-based ones take into account the history of load measurements stored in the speed functions. The FPM-based algorithms define the optimal data distribution through the speed functions. Traditionally, the speed functions are built as piecewise linear functions fitting historic records of the processors' workload. The piecewise linear approximation of the speed functions is used by the geometrical data partitioning algorithm proposed in [1]. It imposes some restrictions on the shape of speed functions but guarantees the existence and uniqueness of the optimal data partitioning.

In this paper, we present a new data partitioning algorithm. This new algorithm formulates the original data partitioning problem as a problem of finding a solution to a multi-dimensional system of nonlinear equations. It employs a numerical multi-dimensional non-linear solver to find the optimal partitioning. It is not that restrictive to the shape of speed functions as the geometrical algorithm and therefore can use more accurate approximations of the real-life speed functions. However, the proposed algorithm does not guarantee a unique solution. In this paper, we propose to interpolate the speed functions by the Akima splines. Passing through all experimental points, the Akime splines form a smooth function of a shape that closely reflects the real performance of the processor.

To demonstrate the advantages of the new FPM-based data partitioning algorithm, we present the experimental results of dynamic load balancing of data-intensive iterative routines on highly heterogeneous computational clusters. In our experiments, the speed functions of the processors are dynamically constructed during the iterations of the routine. Use of the functional performance models allows a computational scientist to utilise the maximum available resources on a given cluster. We demonstrate that our algorithm succeed in balancing the load even in situations when the traditional algorithms fail. We show that the Akima splines provide very accurate approximation of the speed functions.

This paper is structured as follows. In Section 2, we describe traditional piecewise linear approximation of speed functions and the geometrical data partitioning algorithm. In Section 3, we present the new FPM-based data partitioning algorithm based on multidimensional solvers. The algorithm uses the Akima spline interpolation of the speed functions. In Section 4, we demonstrate that this algorithm can successfully balance data-intensive iterative routines for the whole range of problem sizes.

## 2 Traditional Piecewise Linear Approximation of Speed Functions and Geometrical Data Partitioning Algorithm

Functions much more accurately represent the speed of processors than constants [2]. Being application-centric and hardware-specific, functional performance models reflect different aspects of heterogeneity. In this section, we describe a traditional approach to approximate the speed of the processors and the geometrical data partitioning algorithm based on this approach.

Let speeds of $p$ processors be represented by positive continuous functions of problem size $s_1(x),...,s_p(x)$: $s_i(x) = \frac{x}{t_i(x)}$, where $t_i(x)$ is the execution time for processing of $x$ elements on the processor $i$. Speed functions are defined at $[0,n]$, where $n$ is a problem size to partition. The optimal data partition is achieved when all processors execute their work at the same time: $t_1(x_1) \approx ... \approx t_p(x_p)$. This can be expressed as:

$$\frac{x_1}{s_1(x_1)} = ... = \frac{x_p}{s_p(x_p)}, \text{ where } x_1 + x_2 + ... + x_p = n. \quad (1)$$

The integer-value solution of these equations, $d_1,...,d_p$, can be represented geometrically by intersection of the speed functions with a line passing through the origin of the coordinate system (Fig. 2).

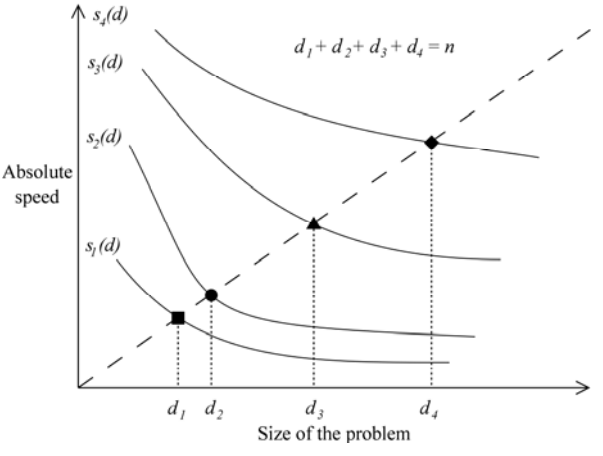



**Fig. 2.** Optimal distribution of computational units showing the geometric proportionality of the number of chunks to the speed of the processor

### 2.1 Data Partitioning Algorithm Based on Geometrical Solution

The geometrical data partitioning algorithm is based on the geometrical solution of the problem (1), assuming that any straight line passing through the origin of the coordinate system intersects the speed functions only once. The algorithm can be summarized as follows. Any line passing through the origin and intersecting all speed functions represents an optimum distribution for a particular problem size. Therefore the space of solutions of the problem (1) consists of all such lines. The two outer bounds of the solution space are selected as the starting point of algorithm. The upper line represents the optimal data distribution $d_1^u,...,d_p^u$ for some problem size $n_u < n$,

$n_u = d_1^u + ... + d_p^u$, while the lower line gives the solution $d_1^l, ..., d_p^l$ for $n_l > n$, $n_l = d_1^l + ... + d_p^l$. The region between two lines is iteratively bisected. At the iteration $k$, the problem size corresponding to the new line intersecting the speed functions at the points $d_1^k, ..., d_p^k$ is calculated as $n_k = d_1^k + ... + d_p^k$. Depending on whether $n_k$ is less than or greater than $n$, this line becomes a new upper or lower bound. Making $n_k$ close to $n$, this algorithm finds the optimal partition of the given problem $d_1, ..., d_p$: $d_1 + ... + d_p = n$. The assumptions about the shape of the speed functions provide the existence and uniqueness of the solution. Fig. 3 illustrates the work of the bisection algorithm.

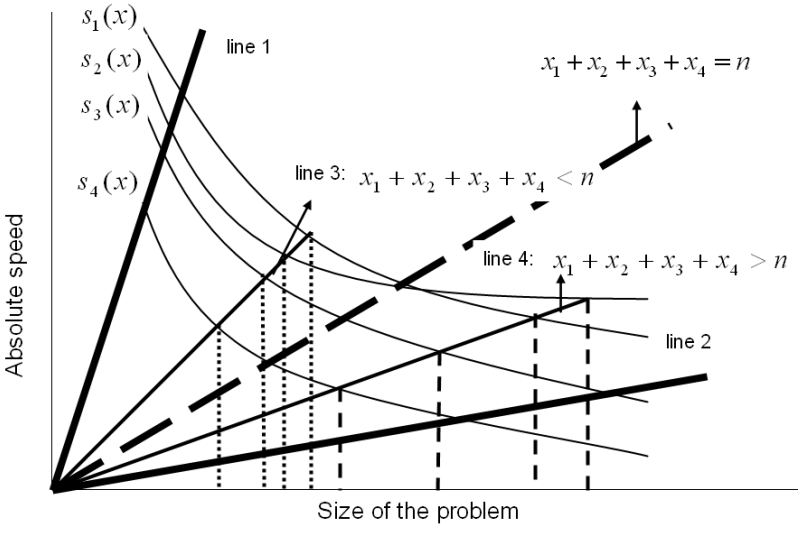



**Fig. 3.** Geometrical data partitioning algorithm. Line 1 (the upper line) and line 2 (the lower line) represent the two initial outer bounds of the solution space. Line 3 represents the first bisection. Line 4 represents the second one. The dashed line represents the optimal solution.

To apply this algorithm to the speed functions, some restrictions are placed on their shape. Experiments performed with many scientific kernels on various heterogeneous networks of workstations have demonstrated that, in general, processor speed could be approximated, within some acceptable degree of accuracy, by a function satisfying the following assumptions [1]:

- On the interval x$[0, X]$, the function is monotonically increasing and concave.
- On the interval $[X, \infty]$, the function is monotonically decreasing.
- Any straight line coming through the origin of the coordinate system intersects the graph of the function in no more than one point.

### 2.2 Piecewise Linear Approximation of Speed Functions

Traditionally, the speed function is built as a piecewise linear function fitting within a band of historic records of workload fluctuations of the processor. To satisfy the above assumptions, the shape of the piecewise linear approximation $s_i(x)$ should be verified after adding the new data point $(d_i^j, s_i^j)$, and the value of $s_i^j$ should be updated when required. Namely, to keep the speed function increasing and convex on the interval $[0, X]$, it is necessary that $\frac{s_i^{j-1} - s_i^{j-2}}{d_i^{j-1} - d_i^{j-2}} > \frac{s_i^j - s_i^{j-1}}{d_i^j - d_i^{j-1}} > \frac{s_i^{j+1} - s_i^j}{d_i^{j+1} - d_i^j} > 0$. This expression represents decreasing tangent of the pieces, which is required for the convex shape of the piecewise linear approximation. On the interval $[X, \infty]$, it is necessary that $s_i^{j-1} \geq s_i^j \geq s_i^{j+1}$ for monotonously decreasing speed function.

The procedure of building piecewise linear approximation is very time consuming, especially for full functional performance models, which are characterized by numerous points. Generating the speed functions is especially expensive in the presence of paging. This forbids building full functional performance models at run time. To reduce the cost of building the speed functions, the partial functional performance models were proposed [3]. They are based on a few points and estimate the real functions in detail only in the relevant regions: $\overline{s}_i(x) \approx s_i(x)$, $1 \leq i \leq p$, $\forall x \in [a, b]$. Both the partial models and the regions are determined at runtime, while the data partitioning algorithm is iteratively applied to the partially built speed functions. The result of the data partitioning, the estimate of the optimal data distribution $d_1^k, ..., d_p^k$, determines the next experimental points $(d_1^k, s_1(d_i^k)), ..., (d_p^k, s_p(d_p^k))$ to be added to the partial models $\overline{s}_1(x), ..., \overline{s}_p(x)$. The more points are added, the closer the partial functions approximate the real speed functions in the relevant regions. Fig. 4 illustrates the construction of the partial speed functions, using the piecewise linear approximation and the geometrical data partitioning algorithm.

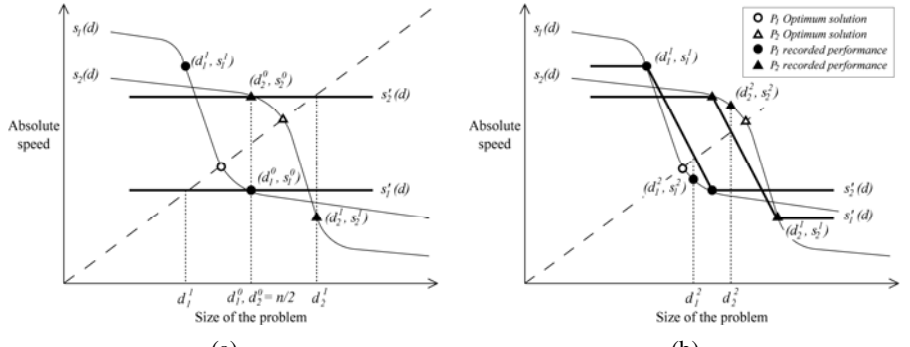



**Fig. 4.** Construction of the partial speed functions, using the piecewise linear approximation and the geometrical data partitioning algorithm

After adding the experimental points to the partial speed functions, their shape is adjusted to satisfy the above assumptions. The low cost of partial building the models makes it ideal for employment in self-adaptive parallel applications, particularly in dynamic load balancing.

## 3 New Data Partitioning Algorithm Based on Multidimensional Root-Finding

The geometrical data partitioning algorithm requires each speed function to be monotonically increasing and concave up to some point and then monotonically decreasing and in addition to be intersected only once by any line passing from the origin. In general, speed functions have this shape, but it is not always the case. For example, a non-optimised algorithm such as Netlib BLAS can have a sawtooth function due to cache misses (Fig. 5). A less accurate function must be fitted to the data to satisfy the shape restrictions (Fig. 5(a)).

Here we present a new data partitioning algorithm which allows us to remove these restrictions and therefore represent the speed of the processor with more accurate continuous functions. This allows us to perform more accurate partitioning. For example, by using the more accurate fit in Fig. 5(b), we can achieve a speedup of 1.26 for some problem sizes. For different routines this speedup could potentially be much bigger.

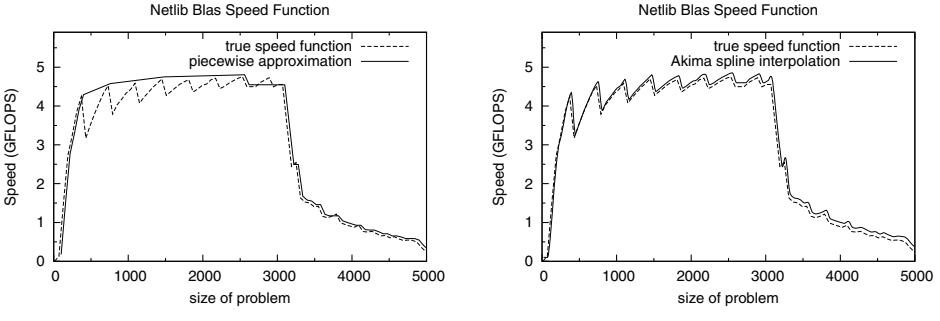



**Fig. 5.** Speed function for non-optimised Netlib BLAS. (a) Fitting shape restricted piecewise approximation. (b) Fitting Akima spline interpolation. Both speed functions have been offset slightly for clarity.

### 3.1 Data Partitioning Algorithm Based on Nonlinear Multidimensional Root Finding

If the processor speeds are approximated by continuous differentiable functions of arbitrary shape, the problem of optimal data partitioning (1) can be formulated as ***multidimensional root finding*** for the system of nonlinear equations $\mathbf{F(x) = 0}$, where

$$\mathbf{F}(\mathbf{x}) = \begin{cases} n - \sum_{i=1}^{p} x_i \\ \frac{x_i}{s_i(x_i)} - \frac{x_1}{s_1(x_1)}, \quad 2 \le i \le p \end{cases} . \tag{2}$$

$\mathbf{x} = (x_1, ..., x_p)$ is a vector of real numbers corresponding a data partition $\mathbf{d} = (d_1, ..., d_p)$. The first equation specifies the distribution of $n$ computational units between $p$ processors. The rest specify the balance of computational load. The problem (2) can be solved by a number of iterative algorithms based on the Newton–Raphson method:

$$\mathbf{x_{k+1}} = \mathbf{x_k} - \mathbf{J(x_k)F(x_k)} . \tag{3}$$

The equal data distribution

$$\mathbf{x}^0 = (n/p, ..., n/p) \tag{4}$$

can be reasonably taken as the initial guess for the location of the root. $\mathbf{J}(\mathbf{x})$ is a Jacobian matrix, which can be calculated as follows:

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} -1 & -1 & ... & -1 \\ -\frac{s_1(x_1) - x_1 s_1'(x_1)}{s_1^2(x_1)} & \frac{s_2(x_2) - x_2 s_2'(x_2)}{s_2^2(x_2)} & 0 & 0 \\ ... & 0 & ... & 0 \\ -\frac{s_1(x_1) - x_1 s_1'(x_1)}{s_1^2(x_1)} & 0 & 0 & \frac{s_p(x_p) - x_p s_p'(x_p)}{s_p^2(x_p)} \end{pmatrix} . \tag{5}$$

We use the HYBRJ algorithm, a modified version of Powell's Hybrid method, implemented in the MINPACK library [4]. It retains the fast convergence of the Newton method and reduces the residual when the Newton method is unreliable. Our experiments demonstrated that for the given vector-function (2) and initial guess (4), the HYBRJ algorithm is able to find the root $\mathbf{x}^* = (x_1^*, ..., x_p^*)$, which will be the optimal data partition after rounding and distribution of remainders: $\mathbf{d} = round(\mathbf{x}^*)$.

### 3.2 New Approximation of Partial Speed Functions Based on the Akima Splines

Let us consider a set of $k$ data points $(x_i, s_i)$, $0 < x_i < n$, $1 \le i \le k$. Here and after in this section, the data points $(x_i, s_i)$ correspond to a single processor, for which the speed function $s(x)$ is approximated. To approximate the speed function in the interval $[0, n]$, we also introduce two extra points: $(0, s_1)$ and $(n, s_k)$. The linear interpolation does not satisfy the condition of differentiability at the breakpoints

$(x_i, s_i)$. The spline interpolations of higher orders have derivatives but may yield significant oscillations in the interpolated function. However, there is a special non-linear spline, the ***Akima spline*** [5], that is stable to outliers (Fig. 6). It requires no less than 5 points. In the inner area $[x_3, x_{k-2}]$, the interpolation error has the order $O(h^2)$. This interpolation method does not require solving large systems of equations and therefore it is computationally efficient.

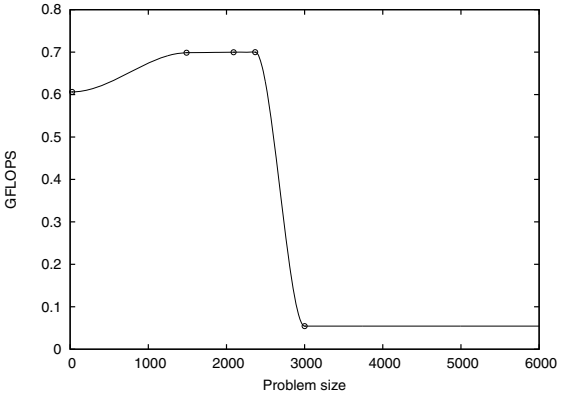



**Fig. 6.** Akima spline interpolation of a dynamically built functional performance model

When the model consists of less than 5 data points, the Akima splines interpolation can be bzuilt for an extended model that duplicates the values of the left- and rightmost points, $s_1, s_k$, as follows:

1. $k = 1$: $x_1 = n/p$, $s_1 = s(n/p)$, the extended model specifies the constant speed as $(0, s_1), \left(\frac{x_1}{2}, s_1\right), (x_1, s_1), \left(\frac{n - x_1}{2}, s_1\right), (n, s_1)$.
2. $k = 2$: the extended model is $(0, s_1), (x_1, s_1), (x_2, s_2), \left(\frac{n - x_2}{2}, s_2\right), (n, s_2)$.
3. $k = 3$: the extended model is $(0, s_1), (x_1, s_1), (x_2, s_2), (x_3, s_3), (n, s_3)$.

**Proposition 1.** The speed functions $s_i$ are defined within the range $0 < x \le n$ and are bounded, continuous, positive, non-zero and have bounded, continuous first derivatives.

**Proof.** The data points $(x_i, s_i)$ are calculated with $s_i(x) = \frac{x}{t_i(x)}$. As it is a practical requirement that each iteration finishes in a finite time and the Akima splines closely fit the data points with continuous smooth functions we can conclude that $s_i$ is continuous, bounded, positive, non-zero within the range $0 < x \le n$. A feature of Akima splines is that they have continuous first derivatives [5].

### 3.3 Convergence and Complexity Analysis

**Proposition 2.** Within the range $0 < x \le n$, the system of nonlinear equations $\mathbf{F(x) = 0}$ contains no stationary points and the functions $f_i\mathbf{(x)}$ have bounded, continuous first derivates, where $f_i\mathbf{(x)}$ is the *i*'th equation of $\mathbf{F(x)}$.

**Proof.** $\mathbf{F'(x)}$ is non-zero for all $0 < x \le n$, hence $\mathbf{F(x)}$ contains no stationary points. $f_0\mathbf{(x)}$ has a constant first derivative. For $f_i\mathbf{(x)}$, $1 \le i < n$, if $s_i$ and $s_i'$ are continuous, bounded and if $s_i$ is non zero then $f_i'\mathbf{(x)}$ is bounded, continuous. This requirement is satisfied by proposition 1.

**Proposition 3.** The new data partitioning algorithm presented in this section converges in a finite number of iterations.

**Proof.** It is proven in [4] that if the range of x is finite, and $\mathbf{F(x)}$ contains no stationary points and if $f_i'\mathbf{(x)}$ is bounded continuous then the HYBRJ solver will converge to $|\mathbf{F(x)}| < \varepsilon$, where $\varepsilon$ is a small positive number, in a finite number of iterations. These requirements are satisfied by proposition 2.

**Proposition 4.** The complexity of one iteration of the solver is $O(p^2)$.

**Proof.** It is show in [6] that the HYBRJ solver has complexity $O(p^2)$. All other steps of the algorithm are of order $O(p)$.

The number of solver iterations depends on the shape of the functions. In practice we found that often 2 iterations are sufficient when the speed functions are very smooth and up to 30 iterations when partitioning in regions of rapidly changing speed functions.

## 4 Dynamic Load Balancing of Parallel Computational Iterative Routines

In this section, we demonstrate how the new data partitioning algorithm can be used for dynamic load balancing of parallel computational iterative routines on highly heterogeneous platforms.

Iterative routines have the following structure: $x^{k+1} = f(x^k)$, $k = 0,1,...$ with $x^0$ given, where each $x^k$ is an $n$-dimensional vector, and $f$ is some function from $\mathbf{R}^n$ into itself [7]. The iterative routine can be parallelized on a cluster of p processors by letting $x^k$ and $f$ be partitioned into $p$ block-components. In an iteration, each processor calculates its assigned elements of $x^{k+1}$. Therefore, each iteration is dependent on the previous one. The performance of computational iterative routines can be represented by the speed of a single iteration as all iterations perform the same amount of computation.

The objective of load balancing algorithms for iterative routines is to distribute computations across a cluster of heterogeneous processors in such a way that all processors will finish their computation within the same time and thereby minimising the overall computation time: $t_i \approx t_j$, $1 \le i, j \le p$. The computation is spread across a cluster of $p$ processors $P_1,...,P_p$ such that $p << n$. Processor $P_i$ contains $d_i$ elements of $x^k$ and $f$, such that $n = \sum_{i=1}^{p} d_i$.

The traditional approach taken for load balancing of data-intensive iterative routines belongs to static/dynamic centralised predicting-the-future algorithms. In these traditional algorithms, computation load is evaluated either in the first few iterations [8] or at each iteration [9] and globally redistributed among the processors. Current load measurements are used for prediction of future performance. When applied to large scientific problems and highly heterogeneous parallel platforms, this strategy may never balance the load, because it uses simplistic models of processors' performance.

Instead of single speed values, we propose more accurate models, namely, partially built functional performance models. At each iteration, we redistribute data with help of the new data partitioning algorithm based on multidimensional root-finding. Our dynamic load balancing can be summarized as follows. At the iteration $k$ of the routine:

1. The data is distributed in accordance with the partition obtained at the previous iteration $\mathbf{d}^k = (d_1^k,...,d_p^k)$. For $k = 0$, the data is distributed evenly: $\mathbf{d}^0 = (n/p,...,n/p)$.
2. The computation is executed and its performance is evaluated at all processors $s_1^k,...,s_p^k$.
3. The new observation points $\left(d_1^k, s_1^k\right),...,\left(d_p^k, s_p^k\right)$ are added to the partial performance models of processors and ***approximations of the speed functions*** $\overline{s}_1(x),...,\overline{s}_p(x)$ are recalculated.
4. ***Data partitioning algorithm*** is applied to the current approximations of the speed functions and returns the refined partition $\mathbf{d}^{k+1}$ for the next iteration.

Since $\overline{s}_i(x) \to s_i(x)$ as $k \to \infty$, $1 \le i \le p$, this procedure adaptively converges to the optimal data distribution $\mathbf{d}^k \to \mathbf{d}^*$.

This dynamic load balancing algorithm was applied to the Jacobi method, which is representative of the class of iterative routines we study. The program was tested successfully on a cluster of 16 processors. For clarity the results presented here are from two configurations of 4 processors, Table 1. The essential difference is that cluster 1 has one processor with 256MB RAM and cluster 2 has two processors with 256MB RAM.

**Table 1.** Specifications of test nodes. Cluster 1 consists of nodes: $P_1$, $P_3$, $P_4$, $P_5$. Cluster 2 consists of nodes: $P_1$, $P_2$, $P_3$, $P_4$.

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| Processor | 3.6 Xeon | 3.0 Xeon | 3.4 P4 | 3.4 Xeon | 3.4 Xeon |
| RAM (MB) | 256 | 256 | 512 | 1024 | 1024 |

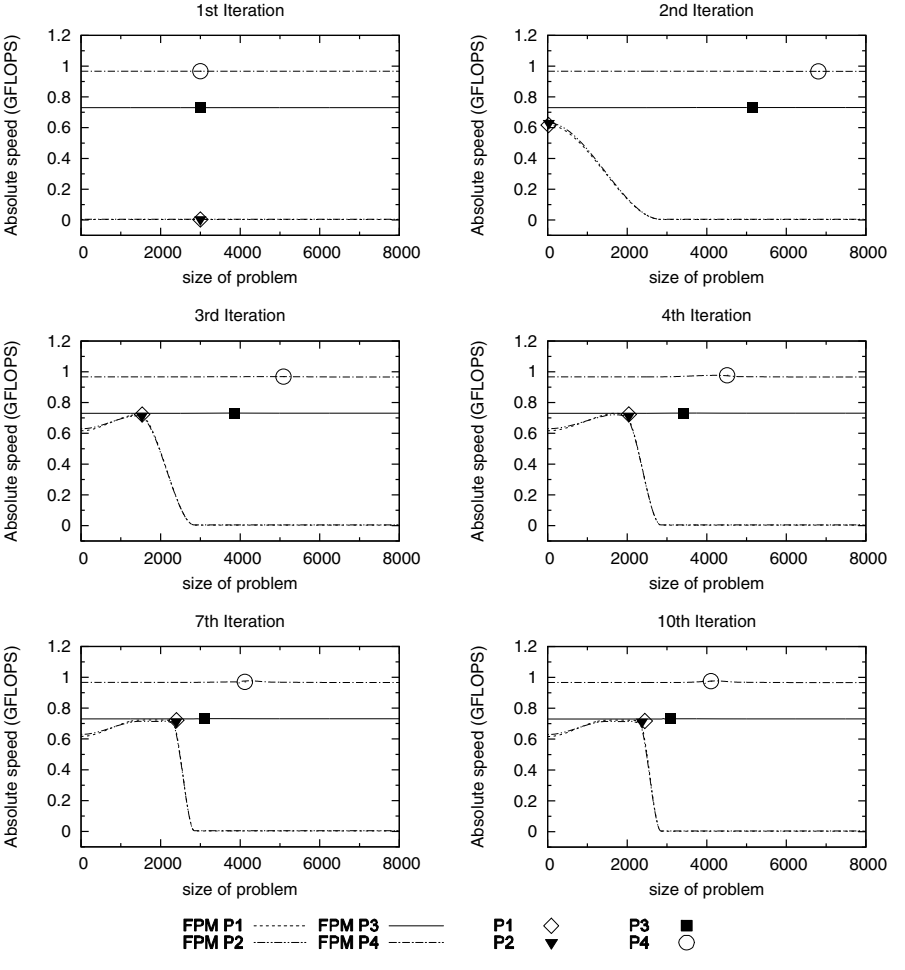



**Fig. 7.** Dynamic load balancing using multidimensional root-finding partitioning algorithm and the Akima spline interpolation for n=12000

The memory requirement of the partitioned routine is a $n \times d_i$ block of a matrix, three $n$ dimensional vectors and some additional arrays of size $p$. For 4 processors with an even distribution, problem sizes of $n$=8000 and $n$=11000 will have a memory

requirement which lies either side of the available memory on the 256MB RAM machines, and hence will be good values for benchmarking.

Fig. 7 illustrates the work of this algorithm for the Jacobi method for 4 processors with $n = 12000$. The algorithm converges to the optimal data distribution with each iteration. By the $7^{th}$ iteration optimum partitioning has been achieved.

Fig. 8 shows the speedup of the CPM and FPM algorithms over a homogeneous distribution. The FPM algorithm used in the experiments is the one based on nonlinear multidimensional root finding. For small problem sizes the speedup is not realised because of the cost involved with data redistribution, however as the size increases both load balancing algorithms improve up to the point were the traditional algorithm based on a constant performance model fails, from which point it performs worse then the homogeneous distribution. The speed up achieved by FPM based load balancing increases as the difference between the relative speeds of the processors increases.

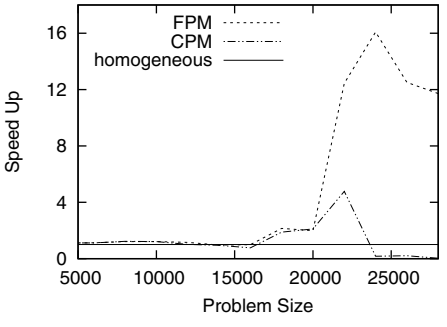



**Fig. 8.** Speed up of dynamic load balancing algorithms over a homogeneous distribution of $n / p$ using a cluster of 16 heterogeneous machines

## 5 Conclusion

Functional performance models of processors are successfully applied to balancing the computational load of data-intensive parallel applications on dedicated highly heterogeneous HPC platforms. Based on a history of time measurements, they better reflect different aspects of heterogeneity of processors. Traditionally, the speed of a processor is approximated by a piecewise linear function. The larger the number of the experimental points, the more accurate the speed function is. However, there is a cost associated with obtaining an experimental data point, which requires execution of a computational kernel for a specified problem size. In addition, the straightforward geometrical solution of the data partitioning problem imposes some restrictions on the shape of speed functions. This requires some adjustments of the experimental points and may result to the non-optimal data partitioning because the speed functions become less accurate. In this paper, we proposed a new accurate approximation of speed functions and a new algorithm that employs the numerical solution of the data partitioning problem. We relax the restrictions on the shape of speed functions and

numerically solve the system of non-linear equations that formalizes the problem of optimal data partitioning. We have shown that the dynamic load balancing algorithms based on functional models can be used successfully with any problem size and on a wide class of heterogeneous platforms.

**Acknowledgment.** This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/IN.1/I2054.

## References

1. Lastovetsky, A., Reddy, R.: Data Partitioning with a Functional Performance Model of Heterogeneous Processors. Int. J. High Perform. Comput. Appl. 21, 76–90 (2007)
2. Lastovetsky, A., Reddy, R., Higgins, R.: Building the Functional Performance Model of a Processor. In: SAC 2006, pp. 746–753. ACM, New York (2006)
3. Lastovetsky, A., Reddy, R.: Distributed Data Partitioning for Heterogeneous Processors Based on Partial Estimation of Their Functional Performance Models. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 91–101. Springer, Heidelberg (2010)
4. Powell, M.J.D.: A Hybrid Method for Nonlinear Equations. In: Gordon, Breach (eds.) Numerical Methods for Nonlinear Algebraic Equations, pp. 87–114 (1970)
5. Akima, H.: A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures. J. ACM 17, 589–602 (1970)
6. Powell, M.J.D.: A Fortran Subroutine for Solving Systems on Nonlinear Algebraic Equations. In: Gordon, Breach (eds.) Numerical Methods for Nonlinear Algebraic Equations, pp. 115–161 (1970)
7. Bahi, J.M., Contassot-Vivier, S., Couturier, R.: Dynamic Load Balancing and Efficient Load Estimators for Asynchronous Iterative Algorithms. IEEE T. Parall. Distr. 16, 289–299 (2005)
8. Martínez, J.A., Garzón, E.M., Plaza, A., García, I.: Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE. J. Supercomput., 1–9 (November 2009)
9. Galindo, I., Almeida, F., Badía-Contelles, J.M.: Dynamic Load Balancing on Dedicated Heterogeneous Systems. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 64–74. Springer, Heidelberg (2008)

# An Efficient Evolutionary Scheduling Algorithm for Parallel Job Model in Grid Environment

Piotr Switalski[1] and Franciszek Seredynski[2,3]

[1] Siedlce University of Natural Sciences and Humanities
Computer Science Department
3 Maja 54, 08-110 Siedlce, Poland
[2] Polish-Japanese Institute of Information Technology
Koszykowa 86, 02-008 Warsaw, Poland
[3] Institute of Computer Science
Polish Academy of Sciences
Ordona 21, 01-237 Warsaw, Poland

**Abstract.** In this paper we propose an efficient parallel job scheduling algorithm for a grid environment. The model implies two stage scheduling. At the first stage, algorithm allocates jobs to the suitable machines, where at the second stage jobs are independently scheduled on each machine. Allocation of jobs on the first stage of the algorithm is performed with use of a relatively new evolutionary algorithm called Generalized Extremal Optimization (GEO). GEO is inspired by a simple coevolutionary model known as Bak-Sneppen model. Scheduling on the second stage is performed by some proposed heuristic. We compare GEO-based scheduling algorithm applied on the first stage with Genetic Algorithm (GA)-based scheduling algorithm. Experimental results show that the GEO, despite of its simplicity, outperforms the GA algorithm in all range of scheduling instances.

**Keywords:** Grid computing, evolutionary algorithm, generalized extremal optimization, parallel job.

## 1 Introduction

Distributed computing has recently become very popular technique to provide high performance computing for computationally intensive applications. This term is often described as "the Grids" or "Grid Computing". In grid computing there are multi-institutional virtual organizations with shared and coordinated resources. As resource we can define direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving [4].

Many studies propose the distributed management system [1] against the centralized scheduling [2]. There are also combination of distributed and centralized management [11]. It can be characterized by a hierarchical multilayer resource management [8]. The first (higher) layer is often assigned to a global grid scheduler. In this layer jobs are scheduled among the machines in the grid. At the

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 347–357, 2011.

second layer (lower) a local management system exists, in which previously assigned jobs are scheduled in the local machine.

The scheduling in a grid system is responsible for resource discovery, resources selection, job assignment and aggregation of group of resources over a decentralized heterogeneous system [7]. The idea of grid computing has forced development of new algorithms of management of a large number of heterogeneous resources. The execution of the user's application must satisfy both job execution constraints and system policies. Scheduling algorithms applied to the traditional multiprocessor systems are often inadequate to grid systems. On the other hand, many algorithms have been adapted to the grid computing [5], [6], [12]. However, there are many open problems in this field, including the consideration of multilayered system in the scheduling process.

This work is related to offline scheduling problem in grid computing and its resolution using meta-heuristic approaches. These algorithms are often used to solve a class of NP-hard problems. The scheduling problem belongs to this class. In this paper we will show the usefulness of meta-heuristic approaches for the design of efficient grid schedulers. We propose a relatively new meta-heuristic called GEO to solve the scheduling problem together with locally applied scheduling heuristic [9].

The paper is organized as follows. In the next section we define the grid model and the scheduling problem. Section 3 presents the concept of GEO algorithm and its application for the scheduling problem. In Section 4 we present GA-based scheduling algorithm. Section 5 describes local scheduling algorithm. Next, in Section 6 we analyze experimental results comparing the use of GEO and GA-based scheduling algorithms. Last section contains conclusions.

## 2 Grid Model and Scheduling

### 2.1 Model

The grid model is defined as follows [10]. A grid system consists of a set of $m$ parallel machines $M_1$, $M_2$, ..., $M_m$. Each machine $M_i$ has $m_i$ identical processors, called also the size of machine $M_i$. Fig. 1(a) shows an example of the grid system. In the grid system there is a set of $n$ jobs $J_1$, $J_2$, ..., $J_n$. A job $J_j$ is described by a triple $(r_j, size_j, t_j)$. The release time $r_j$ can be defined as the earliest time when

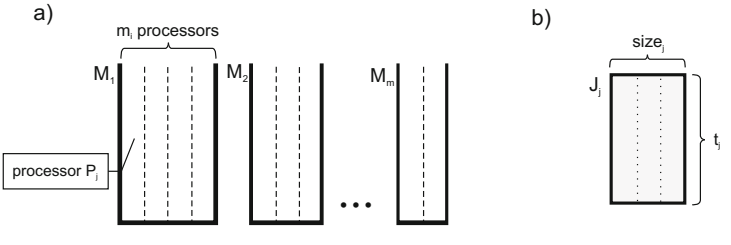



**Fig. 1.** Example of the grid system (a) and the job model (b)

the job can be processed. In this model we assume $r_j >= 0$. A $size_j$ is referred to the processor requirements. It specifies a number of processors required to run the job $J_j$ within assigned machine. We can define this as *degree of parallelism* or a *number of threads*. All threads of the job must be run at the same time and the same machine. The $t_j$ is defined as the execution time of the job $J_j$. Fig. 1(b) shows an example of the job.

Then, the $w_j = t_j * size_j$ denotes the *work* of job $J_j$. A machine execute a job of the $size_j$ when $size_j$ processors are allocated to it during time $t_j$. We assume that job $J_j$ needs to be allocated only within assigned machine $M_i$. In other words, the job cannot be divided into small parts and executed on two or more machines simultaneously. Jobs must not be reallocated from different machines. The machine has to be capable to allocate the job to its processors, so $size_j <= m_i$ must meet.

Let us denote $S$ as a schedule. The completion time of jobs on machine $M_i$ in the schedule $S_i$ is denoted by $C_i(S_i)$. We consider minimization of the time $C_i(S_i)$ on each machine $M_i$ over the system in such a way that the makespan is defined as

$$C_{max} = max_i(C_i(S_i)). \tag{1}$$

The purpose of the scheduling is to distribute jobs among the machines and schedule them to minimize a makespan $C_{max}$.

### 2.2 Two-Stage Scheduling

The proposed scheduling is the two-stage algorithm. An important role of the scheduling process is an appropriate allocation of jobs on machines. We consider system with a different number of processors in machines. It forces to use algorithms which globally distribute jobs among machines of the system. Each job should be allocated in such a way the time of schedule $S$ is minimized.

According to the model we consider the first and the second stages of the scheduling. At the first stage the scheduling algorithm allocates globally jobs to a suitable machines. At each step of the algorithm jobs are reallocated to machines with a lower utilization. The algorithm compares time of schedule with a previous one. Then choose and reallocate the job to another machine, where the time can be minimized. For this stage we choose a meta-heuristic algorithms described in sections 3 and 4.

At the second stage we apply a local scheduling algorithm. It schedules jobs in a particular machine. At this stage we use a heuristic algorithm described in section 5.

## 3 Global Scheduling with Generalized Extremal Optimization Algorithm

### 3.1 The Bak-Sneppen Model and Its Representation in Job Allocation

At the first stage a meta-heuristic algorithm is applied. We propose a relatively new evolutionary algorithm called GEO. The idea of this algorithm is based on

the Bak-Sneppen model [9]. Evolution in this model is driven by a process in which the weakest species in the population, together with its nearest neighbors, is always forced to mutate. The dynamics of this extremal process shows characteristics of Self-Organized Criticality (SOC), such as punctuated equilibrium, that are also observed in natural ecosystems. Punctuated equilibrium is a theory known in evolutionary biology. It states that in evolution there are periods of stability punctuated by a change in an environment that forces relatively rapid adaptation by generating *avalanches*, large catastrophic events that effect the entire system. The probability distribution of these avalanches is described by a power law in the form:

$$p_i = k_i^{-\tau}, \tag{2}$$

where $p_i$ is a probability of mutation of the $i$-th bit (species), $k$ is a position of the $i$-th bit (species) in the ranking, $\tau$ is a positive parameter. If $\tau \to 0$, the algorithm performs a random search, while $\tau \to \infty$, then the algorithm provides deterministic searching. Bak and Sneppen developed a simplified model of an ecosystem in which $N$ species are placed side by side on a line. Fig. 2(a) shows the population of species in the Bak-Sneppen model [9] and the Fig. 2(b) presents the idea of GEO algorithm of the grid scheduling.

In the GEO approach, a population of species is a string of bits that encodes the design variables of the optimization problem, and each bit corresponds to one species. In Fig. 2(a) two variable function $F(x_1, x_2)$ is optimized. Each variable is coded using seven bits, and the whole string - a potential solution of the problem consists of 14 bits (upper part of Fig. 2(a)). Each bit of the string is considered as the species (lower part of Fig. 2(a)) of the Bak-Sneppen model. The number of bits per variable depends on the type of the problem. The population of the GEO algorithm to solve the scheduling problem presented in this paper contains one string of $n$ numbers. The length of the string is equal to the total number of jobs in the grid system. Fig. 2(b) (upper part) shows an example of the GEO string which presents proposed allocation of jobs to machines in the grid. The job numbers are placed in a permutation form. This form defines the order of jobs executing by a local scheduling algorithm. To assign jobs to machines we

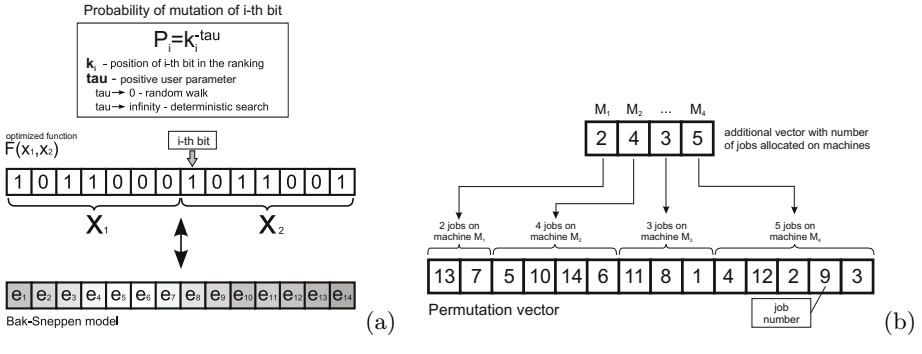



**Fig. 2.** Population of species in the Bak-Sneppen model and its correspondence in the GEO algorithm (a). Representation of grid model in Bak-Sneppen model (b).

set the additional vector (top vector on the Fig. 2(b)) showing the number of jobs allocated to corresponding machines. One can see that, for example, two jobs were allocated to machine $M_1$ and these are jobs 13 and 7. Fig. 2(b) (lower part) shows a relation between coding used in the GEO to solve the scheduling problem and Bak-Sneppen model.

### 3.2 The GEO-Based Scheduling Algorithm

The GEO was originally presented by Sousa and Ramos [9]. In the algorithm each bit (species) is forced to mutate with a probability proportional to the fitness. It is a value associated with a given combination of bits of the GEO string, related to a problem. Change of a single bit of the string results in changing its fitness and indicates the level of adaptability of each bit in the string corresponding to a current solution of a problem. The fitness can gain or loss if a bit is mutated (flipped). After performing a single changing of the string bits and calculating corresponding values of fitness function we can create the sorted ranking of bits by its fitness. Since this moment the probability of mutation $p_i$ of each $i$-th bit placed in the ranking can be calculated by Eq. 2. In Fig 3 we present three types of mutation which can be potentially used by GEO.

In our problem we use integer numbers indicated jobs, instead of bits. Therefore, we propose another type of mutation operator adapted to our problem. We use some operators inspired from technique called Gene Expression Programming (GEP) proposed by Ferreira [3].

The first type of mutation is simply *swap* mutation. In this mutation two jobs are selected. The first job is replaced by the second one and vice versa. The second type of mutation is a *inversion*. First, a job $k$ is selected. Then jobs $k+1$ and $k-1$ are subjected of swap mutation. The last type of mutation is a transposition. In this type of mutation jobs migrate to another machine. In a

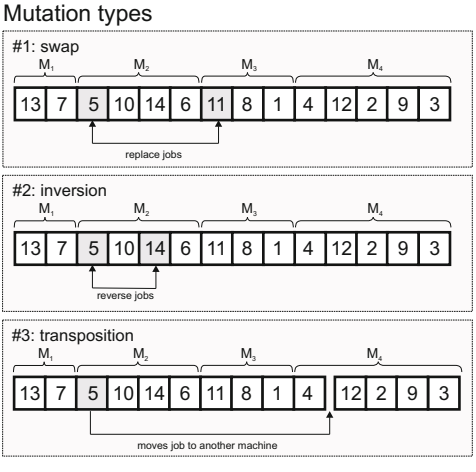



**Fig. 3.** Three types of mutation which can be used by GEO

simple version of this operator the chosen job is moved to machine having the total shortest time of allocated jobs. This type of mutation we used in this work.

The GEO-based scheduling algorithm can be presented as the Algorithm 1.

**Algorithm #1. GEO-based scheduling algorithm**

1. Initialize randomly a permutation string of length $L$ that encodes $n$ jobs in the problem.
2. For the current configuration $C$ of jobs, calculate the value $V$ corresponding to the objective function (1) and set $C_{best} = C$ and $V_{best} = V$.
3. For each job $i$ do
   (a) mutate (transpose) each job and calculate the objective function value $V_i$ of the string configuration $C_i$,
   (b) set the job fitness $F_i$ as $(V_i - R)$, where $R$ is a constant. It serves only as a reference number and can assume any value. The job fitness indicates the relative gain (or loss) that is a result of mutating the job.
   (c) return the string to its previous state.
4. Rank the $N$ jobs according to their fitness values, from $k = 1$ for the least adapted job (the highest ranking) to $k = N$ for the best adapted (the lowest ranking). In a minimization problem higher values of $F_i$ will have higher ranking. If two or more jobs have the same fitness, rank them in random order, but follow the general ranking rule.
5. Choose with an equal probability a job $i$ to mutate according to the probability distribution $p_i = k^{-\tau}$, where $\tau$ is an adjustable parameter. This process called a generation is continued until some job is mutated.
6. Set $C = C_i$ and $V = V_i$.
7. If $F_i < F_{best}$ then set $F_{best} = F_i$ and $C_{best} = C_i$.
8. Repeat steps 3 to 7 until a given stopping criteria is reached.
9. Return $C_{best}$ and $F_{best}$.

## 4 Global Scheduling with GA

GA is a search technique used to find an approximate solution in optimization and search problems. It is a particular class of evolutionary algorithms (EA) that uses mechanisms inspired by evolutionary biology. The algorithm operates on a population of chromosomes coding potential solutions. Chromosomes usually are strings of bits. The fitness function is computed for each individual (chromosome). In the scheduling problem the fitness function is calculated as a makespan $C_{max}$. The Algorithm #2 presents the GA-based scheduling algorithm.

**Algorithm #2. GA-based scheduling algorithm**

1. Choose the initial population of individuals (solutions of the problem)
2. Evaluate the fitness of each individual in the population (calculate the makespan $C_{max}$)

3. Repeat
   (a) Use the Roulette Wheel operator to select individuals for reproduction
   (b) Apply genetic operators: crossover and mutation to generate new solutions
   (c) Evaluate fitness of new individuals
   (d) Replace the population with new individuals
   Until stop condition is satisfied

In our problem strings are permutations of jobs similar to GEO string. As contrasted with GEO, which operates on one string, this algorithm operates on a set of strings.

## 5 Local Scheduling Algorithm

The algorithm of local scheduling allocates jobs within a particular machine. The main idea of this algorithm is to arrange jobs in such a way to minimize the area of empty space of the schedule corresponding to non-busy period processors time. Jobs assigned to a given machine are ordered by the global scheduler (permutated substring). In a local scheduler we use a simple list algorithm. At the beginning jobs $J$ are sorted according to their work $w$. If some jobs have the same size of the work then they are ordered by a sequence given by a global scheduler. Fig. 4 shows an example of local scheduling. It is assumed that a subset (7,1,4,2,6,3,5) of seven jobs was assigned to a machine (see Fig. 4a, upper). We assume that each job consists of a number of threads: job $J_7$ consists of 1 thread, $J_1$ consists of 2 threads, $J_4$ consists of 1 thread, $J_2$ consists of 3 threads, $J_6$ consists of 4 threads, $J_3$ consists of 2 threads and $J_5$ consists of 3 threads. We calculate for each job its work. Let us assume that work for this subset is as follows: $w_7 = 1$,

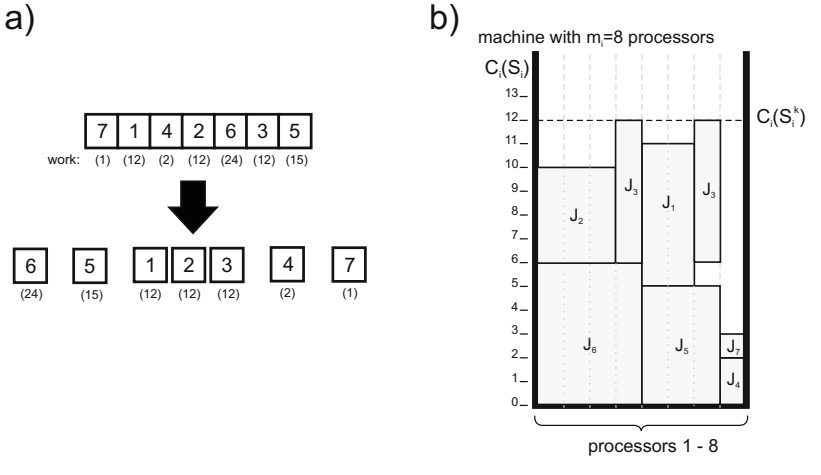



**Fig. 4.** A schema of local scheduling. The representation of the solution and its constructing (a) and allocation in the machine (b).

$w_1 = 12$, $w_4 = 2$, $w_2 = 12$, $w_6 = 24$, $w_3 = 12$, $w_5 = 15$. These jobs are next sorted according to their work $w$ (see Fig. 4a, down). This sorted substring is used directly to assign them to processors of the machine. The job with the largest work is assigned first to processors. Next, the job $J_5$ is assigned to processors, and the remaining jobs are assigned in the following order: $J_1$, $J_2$, $J_3$, $J_4$, $J_7$. In such a way we obtain the schedule $S_i^k$ with the makespan $C_i(S_i^k)$.

## 6 Experimental Results

In this section we compare the performance of GEO and GA-based scheduling algorithms for our scheduling problem defined in this paper. We used in experiments two set of job instances randomly generated. The sets of instances were generated with the following parameters: the number of jobs $N_j$, the average execution time of a job $T_{avg}$, and the average required number of processors $S_{avg}$. It was assumed that the release time $r$ was equal to 0 for both sets.

Calculation of the makespan is the main source of the time complexity of considered algorithms, and the number of evaluations (iterations of the algorithm) of the makespan in both algorithms may be different. We assumed an equal number of evaluations of the makespan for both algorithms.

Initially we show a typical run of GEO and GA algorithm (see Fig. 5). We run these algorithms for 500 jobs set. During the first iterations the GA algorithm quickly improves the solution. After that the algorithm slowly improves solution. The GEO in opposite to the GA, starts with a relatively worse solution. However, it consistently improves solution and quickly finds solution outperforming ones presented by the GA. In the first experiment we use the set of jobs consisting of the number of jobs ($N_j$) ranging from 100 to 1000 jobs. The average execution time $T_{avg}$ was set to 3, average required number of processors $S_{avg}$ was set to 2. For this instance 4 and 8 processor machines were used.

In the GEO we used the following parameters in the experiment: $\tau = 4.0$, a number of iterations of the algorithm $N_{itGEO} = 500$. For GA we set a population size $P_{GA} = 200$, mutation probability $p_m = 0.1$, crossover probability $p_c = 0.75$. The number of iterations was set in range $N_{itGA} \in \{250..2500\}$ (for equal number of the evaluation of the makespan in both algorithms). We repeated each instance of the experiment 20 times.

Tab. 1 shows the results of the experiment. We start from small instances of the problem. In the table we present the minimal time (makespan) obtained by algorithms, and in the brackets the standard deviation calculated for 20 runs of the algorithm. One can see that for 100 jobs the results are similar for both algorithms, but GEO slightly outperforms GA. The standard deviation for the GA is higher than for the GEO. This is visible especially for a small machine sizes. For instances with use of 1000 jobs the standard deviation is smaller for GA than for GEO, however, the makespan is significantly smaller for GEO. This algorithm search the solution more permanently and precisely.

In the second experiment we used the same number of jobs with the average time $T_{avg}$ set to 5, average required number of processors $S_{avg}$ set to 4. Machines

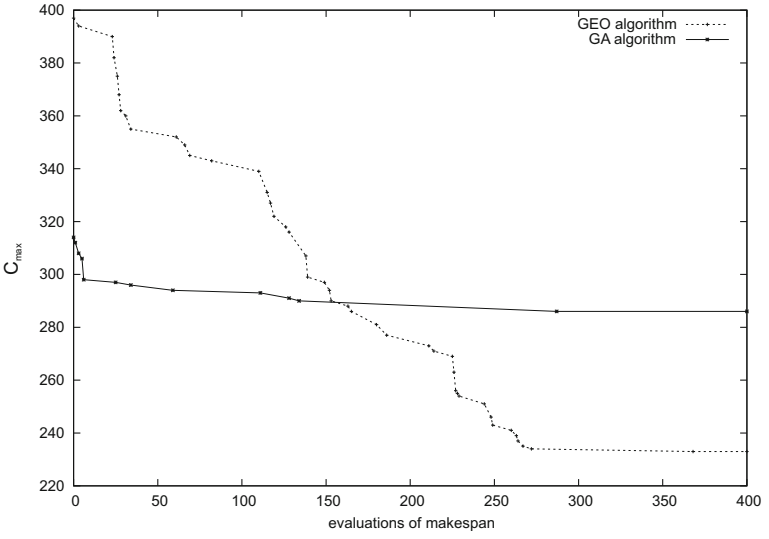



**Fig. 5.** Typical run of GEO and GA algorithm for 500 jobs set ($T_{avg}$=5, $S_{avg} = 4$) in 4 (with 8 and 16 processors) machines environment

**Table 1.** Comparison of the makespan obtained by algorithms: GEO and GA for machines with use of 4 and 8 processors. The model of jobs assumes the average time $T_{avg}$ equal to 3 and average threads $S_{avg}$ equal to 2. In rounded brackets is given the standard deviation.

| | **4 machines** *4_machines_4_8* | | **8 machines** *8_machines_4_8* | | **16 machines** *16_machines_4_8* | | **32 machines** *32_machines_4_8* | |
|---|---|---|---|---|---|---|---|---|
| **Number of jobs** | GEO | GA | GEO | GA | GEO | GA | GEO | GA |
| 100 jobs | 35 (0.3) | 36 (5.7) | 20 (0.4) | 22 (2.1) | 11 (0.6) | 13 (1.3) | 8 (0.5) | 9 (0.8) |
| 500 jobs | 147 (0.4) | 181 (9.2) | 81 (0.4) | 96 (5.0) | 39 (0.6) | 49 (2.7) | 22 (1.4) | 30 (2.0) |
| 1000 jobs | 317 (12.2) | 408 (17.0) | 174 (8.9) | 220 (7.4) | 82 (10.0) | 112 (4.4) | 46 (5.0) | 53 (4.1) |

contained from 8 to 16 processors. The parameter of the GEO and GA algorithms remain the same like in the previous experiment.

The results are presented in the Tab. 2. One can see that the results are similar: the GEO algorithm significantly outperforms the GA algorithm. In the instance with use of 1000 jobs this parameter reaches 24.6 for GEO. This instance especially with use of 4 machines is the most complicated of all. Jobs needs to be allocated in a small number of machines, so set their order and assign to appropriate machine is significant. As we can notice, GEO can find appreciably better outcomes than GA. Explanation of these results are consequence of behavior GEO and GA algorithms. GEO is finding the solution by calculating the makespan for each job from population and accepting population for which mutation of job gives the best result. In GA we calculate fitness function for a

**Table 2.** Comparison of the makespan obtained by algorithms: GEO and GA for machines with use of 8 and 16 processors. The model of jobs assumes the average time $T_{avg}$ equal to 5 and average threads $S_{avg}$ equal to 4. In rounded brackets is given the standard deviation.

| | **4 machines** *4_machines_8_16* | | **8 machines** *8_machines_8_16* | | **16 machines** *16_machines_8_16* | | **32 machines** *32_machines_8_16* | |
|---|---|---|---|---|---|---|---|---|
| **Number of jobs** | GEO | GA | GEO | GA | GEO | GA | GEO | GA |
| 100 jobs | 49 (0.6) | 53 (5.1) | 27 (0.5) | 27 (3.9) | 16 (0.4) | 18 (1.3) | 11 (0.6) | 14 (1.0) |
| 500 jobs | 231 (0.6) | 273 (18.3) | 118 (0.8) | 147 (5.8) | 62 (2.5) | 81 (5.8) | 36 (6.2) | 49 (2.5) |
| 1000 jobs | 467 (24.6) | 597 (16.6) | 238 (16.9) | 305 (11.5) | 132 (10.6) | 165 (6.7) | 72 (11.4) | 94 (2.8) |

population of individuals. In GA mutation only maintain genetic diversity from one generation of a population to the next. Mutation in GEO is more meaningful. GEO is finding a solution more consequently by precisely valuation of jobs and choosing the most suitable solution in the current step of the algorithm.

## 7 Conclusions

In this paper we have proposed a two-stage scheduling algorithm, where we used the relatively new meta-heuristic called GEO to solve the scheduling problem. We compared our results obtained with use of GA. We have shown that GEO-based scheduling algorithm outperforms GA-based scheduling algorithm in term of the makespan. Our future research plans will be oriented on study local scheduling heuristics proposed in the paper.

## References

1. Ernemann, C., Yahyapour, R.: Applying Economic Scheduling Methods to Grid Environments. In: Grid Resource Management - State of the Art and Future Trends, pp. 491–506. Kluwer Academic Publishers, Dordrecht (2003)
2. Ernemann, C., Hamscher, V., Schwiegelshohn, U., Streit, A., Yahyapour, R.: On Advantages of Grid Computing for Parallel Job Scheduling. In: Proceedings of 2nd IEEE International Symposium on Cluster Computing and the Grid, pp. 39–46 (2002)
3. Ferreira, C.: Gene Expression Programming: A New Adaptive Algorithm for Solving Problems Complex Systems, 13(2),87–129 (2001)
4. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal Supercomputer Applications 15(3) (2001)
5. Ghafoor, A., Yang, J.: A Distributed Heterogeneous Supercomputing Management System. Computer 26(6), 78–86 (1993)
6. Hall, R., Rosenberg, A.L., Venkataramani, A.: A Comparison of Dag-Scheduling Strategies for Internet-Based Computing. In: IPDPS 2007 IEEE International Parallel and Distributed Processing Symposium, p. 55 (2007)

7. Murugesan, G., Chellappan, C.: An Economic Allocation of Resources for Multiple Grid Applications. In: Proceedings of the World Congress on Engineering and Computer Science 2009, San Francisco, USA, vol. I, pp. 20–22 (2009)
8. Schwiegelshohn, U.: An Owner-centric Metric for the Evaluation of Online Job Schedules. In: Proceedings of the 2009 Multidisciplinary International Conference on Scheduling: Theory and Applications, pp. 557–569 (2009)
9. Sousa, F.L., Ramos, F.M., Galski, R.L., Muraoka, I.: Generalized Extremal Optimization: A New Meta-heuristic Inspired by a Model of Natural Evolution. Generalized Extremal Optimization: A New Meta-heuristic Inspired by a Model of Natural Evolution, 41–60 (2004)
10. Tchernykh, A., Schwiegelshohn, U., Yahyapour, R., Kuzjurin, N.: On-line hierarchical job scheduling on grids with admissible allocation. Journal of Scheduling 13(5), 545–552 (2010)
11. Vazquez-Poletti, J.L., Huedo, E., Montero, R.S., Llorente, I.M.: A comparison between two grid scheduling philosophies: EGEE WMS and Grid Way. Journal Multiagent and Grid Systems 3(4), 429–440 (2007)
12. Xhafa, F., Alba, E., Dorronsoro, B., Duran, B.: Efficient Batch Job Scheduling in Grids using Cellular Memetic Algorithms. Journal of Mathematical Modelling and Algorithms 7(2), 217–236 (2008)
13. Xhafa, F., Abraham, A. (eds.): Meta-heuristics for Grid Scheduling Problems in Distributed Computing Environments. SCI, vol. 146, pp. 1–37. Springer, Heidelberg (2008)

# On Mapping Graphs of Parallel Programs onto Graphs of Distributed Computer Systems by Recurrent Neural Networks

Mikhail S. Tarkov

A.V. Rzhanov's Institute of Semiconductor Physics,
Siberian branch, Russian Academy of Sciences,
13, Lavrentieva avenue, Novosibirsk, 630090, Russia
tarkov@isp.nsc.ru

**Abstract.** A problem of mapping graphs of parallel programs onto graphs of distributed computer systems by recurrent neural network is formulated. Parameter values providing absence of incorrect solutions are experimentally determined. Optimal solutions are found for mapping a "line"-graph onto a two-dimensional torus due to introduction into Lyapunov function of penalty coefficients for the program graph edges not-mapped onto the system graph edges. For increasing probability of finding optimal mapping, a method for splitting the mapping is proposed. The method essence is a reducing solution matrix to a block-diagonal form. The Wang recurrent neural network is used to exclude incorrect solutions of the problem of mapping the line-graph onto three-dimensional torus. This network converges quicker than the Hopfield one.

**Keywords:** Mapping, graphs of parallel programs, distributed computer systems, neuron, Hopfield network, Wang recurrent network.

## 1 Introduction

A distributed computer system (CS) [1] is a set of elementary computers (ECs) connected by a network that is program-controlled from these computers. Each EC includes a computer module (CM) and a system unit (message router). The message router operates under CM control and has input and output ports connected to the output and input ports of the neighboring ECs, correspondingly. The CS structure is described by the graph $G_s\left(V_s, E_s\right)$, where $V_s$ is the set of ECs and $E_s = V_s \times V_s$ is the set of connections between the ECs.

For distributed CSs, the graph of a parallel program $G_p\left(V_p, E_p\right)$ is usually determined as a set $V_p$ of the program branches (virtual elementary computers) interacting with each other by the point–to–point principle through transferring messages via logical (virtual) channels (which may be unidirectional or bidirectional) of the set $E_p = V_p \times V_p$. Interactions between the processing modules are ordered in time and regular in space for most parallel applications (line, ring, mesh, etc.) (Fig. 1).

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 358–367, 2011.

For this reason, the maximum efficiency of information interactions in advanced high-performance CSs is obtained by using regular graphs $G_s(V_s, E_s)$ of connections between individual computers (hypercube, 2D-torus, or 3D-torus) [2, 3]. The hypercube structure is described by a graph known as a *m*-dimensional Boolean cube with a number of nodes $n = 2^m$. Toroidal structures are *m*-dimensional Euclidean meshes with closed boundaries. The group of automorphisms $E_m$ of such a structure is a direct product of cyclic subgroups $C_{N_k} : E_m = \bigotimes_{k=1}^{m} C_{N_k}$, where $N_k$ is the order of the subgroup and $\otimes$ is the symbol of the direct product. For *m* = 2, we obtain a two-dimensional torus (2D-torus) (Fig. 2); for *m* = 3, we obtain a 3D-torus.

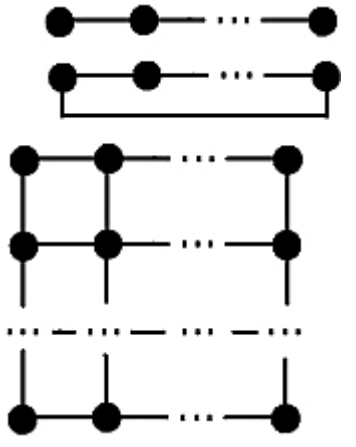


**Fig. 1.** Typical graphs of parallel programs (line, ring and mesh)

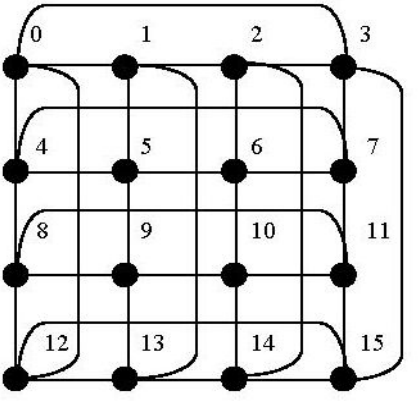



**Fig. 2.** Example of a 2D-torus

In this paper, we consider a problem for mapping graph $G_p\left(V_p, E_p\right)$ of a parallel program onto graph $G_s\left(V_s, E_s\right)$ of a distributed CS, where $n = \left|V_p\right| = \left|V_s\right|$ is a number of program branches (of ECs). The mapping objective is to map nodes of the program graph $G_p$ onto nodes of the system graph $G_s$ one-to-one to carry out mapping $G_p$ edges onto edges of $G_s$ (to establish an isomorphism between the program graph $G_p$ and a spanning subgraph of the system graph $G_s$).

Massive parallelism of data processing in neural networks allows us to consider neural networks as a perspective, high-performance, and reliable tool for solution of complicated optimization problems. The recurrent neural networks [4-10] are a most interesting tool for solution of discrete optimization problems. A model of a globally converged recurrent Hopfield neural network is in good accordance with Dijkstra's self-stabilization paradigm [11]. This signifies that the mappings of parallel program graphs onto graphs of distributed computer systems, carried out by Hopfield networks, are self-stabilizing. An importance of usage of the self-stabilizing mappings is caused by a possibility of breaking the CS graph regularity by failures of ECs and intercomputer connections.

## 2 Hopfield Network for the Mapping Problem

Let us consider a matrix $v$ of neurons with size $n \times n$, each row of the matrix corresponds to some branch of a parallel program and every column of the matrix corresponds to some EC. Each row and every column of the matrix $v$ must contain only one nonzero entry equal to one, other entries must be equal to zero. The energy of the corresponding neural Hopfield network is described by the Lyapunov function

$$\begin{aligned} L &= C \cdot L_c + D \cdot L_d, \\ L_c &= \frac{1}{2}\sum_x \left( \sum_j v_{xj} - 1 \right)^2 + \sum_i \left( \sum_y v_{yi} - 1 \right)^2, \\ L_d &= \frac{1}{2}\sum_x \sum_i \sum_{y \in Nb_p(x)} \sum_{j \neq i} v_{xi} v_{yj} d_{ij}. \end{aligned} \tag{1}$$

Here $v_{xi}$ is a state of the neuron in the row $x$ and column $i$ of the matrix $v$, $C$ and $D$ are parameters of the Lyapunov function. $L_c$ is minimal when each row and every column of $v$ contains only one unity entry (all other entries are zero). Such matrix $v$ is a correct solution of the mapping problem. The minimum of $L_d$ provides minimum of the sum of distances between adjacent $G_p$ nodes mapped onto nodes of the system graph $G_s$. Here $d_{ij}$ is a distance between nodes $i$ and $j$ of the system

graph corresponding to adjacent nodes of the program graph (a "dilation" of the edge of the program graph on the system graph), $Nb_p(x)$ is a neighborhood of the node $x$ on the program graph.

The Hopfield network minimizing the function (1) is described by the equation [5]

$$\frac{\partial u_{xi}}{\partial t} = -\frac{\partial L}{\partial v_{xi}} \tag{2}$$

where $u_{xi}$ is an activation of the neuron with indices $x, i$ $(x, i = 1, ..., n)$,

$$v_{xi} = \frac{1}{1 + \exp(-\beta u_{xi})}$$

is the neuron state (output signal), $\beta$ is the activation parameter. From (1) and (2) we have

$$\frac{\partial u_{xi}}{\partial t} = -C\left(\sum_j v_{xj} + \sum_y v_{yi} - 2\right) - D \sum_{y \in Nb_p(x)} \sum_{j \neq i} v_{yj} d_{ij}. \tag{3}$$

A difference approximation of Equation (3) yields

$$u_{xi}^{t+1} = u_{xi}^t - \Delta t \cdot \left[ C\left(\sum_j v_{xj} + \sum_y v_{yi} - 2\right) + D \sum_{y \in Nb_p(x)} \sum_{j \neq i} v_{yj} d_{ij} \right], \tag{4}$$

where $\Delta t$ is a temporal step. Initial values $u_{xi}^0$ $(x, i = 1, ..., n)$ are stated randomly.

A choice of parameters $\beta, \Delta t, C, D$ [6-10] determines a quality of the solution $v$ of Equation (4). In [8] a dependence between parameters $C$ and $D$ is determined. For the problem (1)-(4)

$$C \approx 100 \cdot D. \tag{5}$$

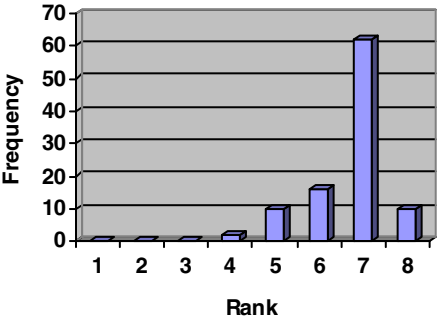
From (4) and (5) it follows that the parameters $\Delta t$ and $D$ are equally influenced on the solution of the equation (4). Therefore we state $\Delta t = 1$ and get

$$u_{xi}^{t+1} = u_{xi}^t - C \cdot \left(\sum_j v_{xj} + \sum_y v_{yi} - 2\right) - D \cdot \sum_{y \in Nb_p(x)} \sum_{j \neq i} v_{yj} d_{ij}. \tag{6}$$

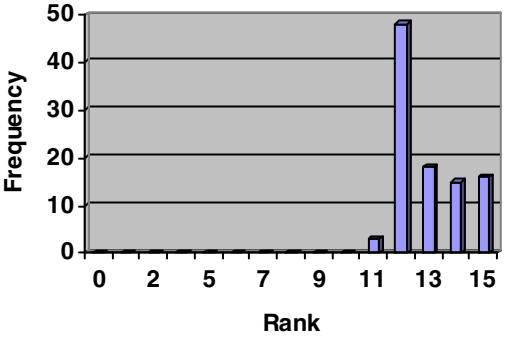
Let $\beta = 0.1$ (this value was stated in [10]). We will try to choose the value $D$ to provide the absence of incorrect solutions.

## 3 Mapping by the Hopfield Network

Let us evaluate the mapping quality by a number of coincidences of the program edges with edges of the system graph. We call this number a mapping rank. The mapping rank is an approximate evaluation of the mapping quality because the mappings with different dilations of the edges of the program graph may have the same mapping rank. Nevertheless, the maximum rank value, which equals to the number $|E_p|$ of edges of the program graph, corresponds to optimal mapping, i.e. to a global minimum of $L_d$ in (1). Our objective is to determine the mapping algorithm parameters providing maximum probability of the optimal mapping. As an example for investigation of the mapping algorithm we consider the mapping of a line-type program graph onto a 2D-torus. Maximal value of the mapping rank for a line with $n$ nodes is obviously equal to $n-1$.




a) $n=9$




b) $n=16$

**Fig. 3.** Histograms of mappings for the neural network (6)

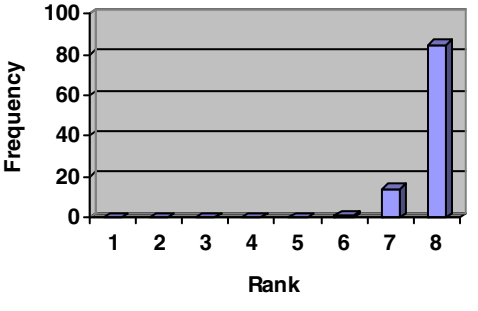
For experimental investigation of the mapping quality, the histograms of the mapping rank frequencies are used for a number of experiments equals to 100. The experiments for mapping the line onto the 2D-torus with the number of nodes $n = l^2$, $l = 3, 4,$ where $l$ is the cyclic subgroup order, are carried out.

For $D \geq 8$ the correct solutions are obtained for $n = 9$ and $n = 16$, but as follows from Fig. 3a and Fig. 3b for $D = 8$, the number of solutions with optimal mapping, corresponding to the maximal mapping rank, is small.

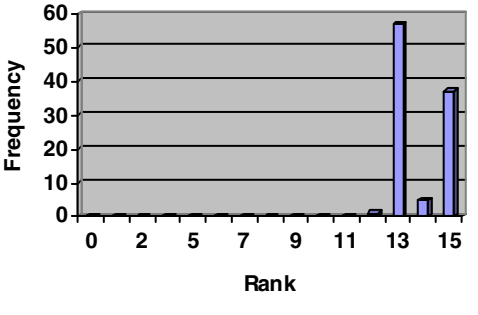
To increase the frequency of optimal solutions of Equation (6) we replace the distance values $d_{ij}$ by the values

$$c_{ij} = \begin{cases} d_{ij} & d_{ij} = 1 \\ p \cdot d_{ij} & d_{ij} > 1 \end{cases} \tag{7}$$

where $p$ is a penalty coefficient for the distance $d_{ij}$ exceeding the value 1, i.e. for non-coincidence of the edge of the program graph with the edge of the system graph. So, we obtain the equation




a) $n = 9$




b) $n = 16$

**Fig. 4.** Histograms of mappings for the neural network (8)

$$u_{xi}^{t+1} = u_{xi}^{t} - C \cdot \left( \sum_{j} v_{xj} + \sum_{y} v_{yi} - 2 \right) - D \cdot \sum_{y \in Nb_p(x)} \sum_{j \neq i} v_{yj} c_{ij}. \quad (8)$$

For the above mappings with $p = n$ we obtain the histograms shown on Fig. 4a and Fig. 4b. These histograms indicate the improvement of the mapping quality but for $n = 16$ the suboptimal solutions with the rank 13 have maximal frequency.

## 4 Splitting Method

To decrease a number of local extremums of Function (1), we partition the set $\{1, 2, ..., n\}$ of subscripts $x$ and $i$ of the variables $v_{xi}$ to $K$ sets $I_k = \{(k-1)q,\ (k-1)q+1, ..., k \cdot q\}$, $q = n / K$, $k = 1, 2, ..., K$, and map the subscripts $x \in I_k$ only to the subscripts $i \in I_k$, i.e. we reduce the solution matrix $v$ to a block-diagonal form. Then taking into account Expression (7), the Lyapunov function (1) is transformed into

$$L = \frac{C}{2} \sum_{k=1}^{K} \left[ \sum_{x \in I_k} \left( \sum_{j \in I_k} v_{xj} - 1 \right)^2 + \sum_{i \in I_k} \left( \sum_{y \in I_k} v_{yi} - 1 \right)^2 \right] +$$

$$+ \frac{D}{2} \sum_{k=1}^{K} \sum_{x \in I_k} \sum_{i \in I_k} \sum_{y \in Nb_p(x)} \sum_{j \neq i} v_{xi} v_{yj} c_{ij},$$

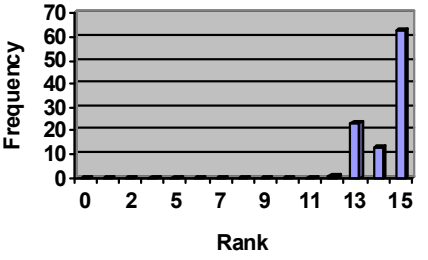
and the Hopfield network is described by the equation

$$u_{xi}^{t+1} = u_{xi}^{t} - C \cdot \left( \sum_{j \in I_k} v_{xj} + \sum_{y \in I_k} v_{yi} - 2 \right) - D \cdot \sum_{y \in Nb_p(x)} \sum_{j \neq i} v_{yj} c_{ij},$$

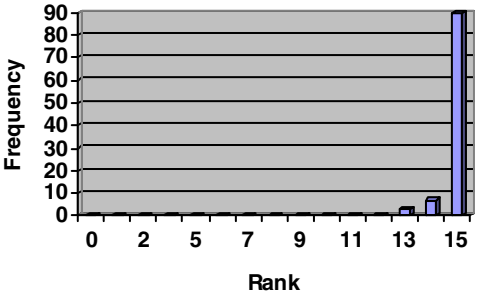
$$v_{xi} = \frac{1}{1 + \exp(-\beta u_{xi})}, \quad x, i \in I_k, k = 1, 2, ..., K. \quad (9)$$

In this case $v_{xi} = 0$ for $x \in I_k, i \notin I_k, k = 1, 2, ..., n.$

In this approach which we call a splitting, for mapping line with the number of nodes $n = 16$ onto 2D-torus, we have for $K = 2$ the histogram presented on Fig. 5a. From Fig. 4b and Fig. 5a we see that the splitting method essentially increases the frequency of optimal mappings. The increase of the parameter $D$ up to the value $D = 32$ results in additional increase of the frequency of optimal mappings (Fig. 5b).




a) $n=16$, $K=2$, $D=8$.




b) $n=16$, $K=2$, $D=32$.

**Fig. 5.** Histograms of mappings for the neural network (9)

## 5 Mapping by the Wang Network

In a recurrent Wang neural network [6] $L_d$ in Expression (1) is multiplied by the value $\exp\left(-t/\tau\right)$ where $\tau$ is a parameter. For the Wang network Equation (9) is reduced to

$$u_{xi}^{t+1}=u_{xi}^{t}-C\cdot\left(\sum_{j\in I_k}v_{xj}+\sum_{y\in I_k}v_{yi}-2\right)-D\cdot\sum_{y\in Nb_p(x)}\sum_{j\neq i}v_{yj}c_{ij}\exp\left(-t/\tau\right),$$
$$v_{xi}=\frac{1}{1+\exp\left(-\beta u_{xi}\right)},\quad x,i\in I_k,k=1,2,...,K. \tag{10}$$

We note that in experiments we frequently have incorrect solutions if for a given maximal number of iterations $t_{\max}$ (for example, $t_{\max}=10000$) the condition of convergence $\sum_{x,i}\left|u_{xi}^{t}-u_{xi}^{t-1}\right|<\varepsilon$, $\varepsilon=0.01$ is not satisfied. The introduction of

factor $\exp\left(-t/\tau\right)$ accelerates the convergence of the recurrent neural network and the number of incorrect solutions is reduced.

So, for the three-dimensional torus with $n = 3^3 = 27$ nodes and $p = n,\ K = 3, D = 4096, \beta = 0.1$ in 100 experiments we have the following results:

1) On the Hopfield network we have 23 incorrect solutions, 43 solutions with Rank 25 and 34 optimal solutions (with Rank 26).

2) On the Wang network with the same parameters and $\tau = 500$ we have all (100) correct solutions, where 27 solutions have Rank 25 and 73 solutions are optimal (with Rank 26).

## 6 Conclusion

A problem of mapping graphs of parallel programs onto graphs of distributed computer systems by recurrent neural networks is formulated. The parameter values providing the absence of incorrect solutions are experimentally determined.

A penalty parameter is introduced into the Lyapunov function of the Hopfield network for the program graph edges not-mapped onto the edges of the system graph. As a result, we obtain optimal solutions for mapping a line-type graph of parallel program onto two-dimensional torus with the same number of nodes $n \in \{9, 16\}$. To increase the probability (the frequency) of optimal mappings, we propose to use:

1) a splitting method reducing the solution matrix to a block-diagonal form;

2) the Wang recurrent network which converges more rapidly than the Hopfield network.

As a result we have high frequency of optimal solutions (for 100 experiments):

1) more than 80% for the two-dimensional tori ( $n = 3^2 = 9$ and $n = 4^2 = 16$ );
2) more than 70% for three-dimensional torus $(n = 3^3 = 27)$.

Further investigations must be directed to increasing the probability of getting optimal solutions of the mapping problem when the number of the parallel program nodes is increased.

## References

1. Tarkov, M.S.: Mapping Parallel Program Structures onto Structures of Distributed Computer Systems. Optoelectronics, Instrumentation and Data Processing 39(3), 72–83 (2003)
2. Cray T3E, http://www.cray.com/products/systems/crayt3e/1200e.html
3. Yu, H., Chung, I.H., Moreira, J.: Topology Mapping for Blue Gene/L Supercomputer. In: SC 2006: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, pp. 52–64. ACM Press, New York (2006)

4. Hopfield, J.J., Tank, D.W.: Neural Computation of Decisions in Optimization Problems. Biological Cybernetics 52, 141–152 (1985)
5. Takefuji, Y., Lee, K.C.: Artificial Neural Networks for Four-Coloring Map Problems and K-Colorability Problems. IEEE Trans. Circuits Syst. 38(3), 326–333 (1991)
6. Wang, J.: Analysis and Design of a Recurrent Neural Network for Linear Programming. IEEE Trans. On Circuits and Systems-I: Fundamental Theory and Applications 40(9), 613–618 (1993)
7. Smith, K.A.: Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research. INFORMS Journal on Computing 11(1), 15–34 (1999)
8. Feng, G., Douligeris, C.: The Convergence and Parameter Relationship for Discrete-Time Continuous-State Hopfield Networks. In: IEEE IJCNN 1999, Washington DC, pp. 376–381 (1999)
9. Tarkov, M.S., Lapukhov, S.A.: Mapping a Parallel Program Structure onto Distributed Computer System Structure by the Hopfield Neural Network with Fuzzy Scheduling Parameters. Bulletin of the Novosibirsk Computing Center, Comp. Science 19, 83–88 (2003)
10. Tarkov, M.S.: Construction of Hamiltonian Cycles by Recurrent Neural Networks in Graphs of Distributed Computer Systems. Numerical Analysis and Applications 3(4), 381–388 (2010)
11. Dijkstra, E.W.: Self-stabilizing Systems in Spite of Distributed Control. Commun. ACM 17(11), 643–644 (1974)

# Slot Selection and Co-allocation for Economic Scheduling in Distributed Computing

Victor Toporkov[1], Alexander Bobchenkov[1], Anna Toporkova[2], Alexey Tselishchev[3], and Dmitry Yemelyanov[1]

[1] Computer Science Department, Moscow Power Engineering Institute, ul. Krasnokazarmennaya, 14, Moscow, 111250, Russia
{ToporkovVV,BobchenkovAV,YemelyanovDM}@mpei.ru
[2] Moscow State Institute of Electronics and Mathematics, Bolshoy Trekhsvyatitelsky per., 1-3/12, Moscow, 109028, Russia
annastan@mail.ru
[3] European Organization for Nuclear Research (CERN), Geneva, 23, 1211, Switzerland
Alexey.Tselishchev@cern.ch

**Abstract.** In this paper, we present slot selection algorithms for job batch scheduling in distributed computing with non-dedicated resources. Jobs are parallel applications and these applications are independent. Existing approaches towards resource co-allocation and job scheduling in economic models of distributed computing are based on search of time-slots in resource occupancy schedules. A launch of a parallel job requires a co-allocation of a specified number of slots. The sought time-slots must match requirements of necessary span, computational resource properties, and cost. Usually such scheduling methods consider only one suited variant of time-slot set. This paper discloses a scheduling scheme that features multi-variant search. Two algorithms of linear complexity for search of alternative variants are proposed. Having several optional resource configurations for each job makes an opportunity to perform an optimization of execution of the whole batch of jobs and to increase overall efficiency of scheduling.

**Keywords:** Scheduling, co-allocation, slot, resource request, job, batch, task..

## 1 Introduction

Economic models for resource management and scheduling are very effective in distributed computing with non-dedicated resources, including Grid [1, 2], utility computing [3], cloud computing [4], and multiagent systems [5]. There is a good overview of some approaches to forming of different deadline and budget constrained strategies of economic scheduling in [6]. In [7] heuristic algorithms for slot selection based on user defined utility functions are introduced.

While implementing economic policy, resource brokers usually optimize the performance of a specific application [1, 6, 7] in accordance with the application-level scheduling concept [8]. When establishing virtual organizations (VO), the optimization is performed for the job-flow scheduling [9, 10]. Corresponding

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 368–383, 2011.

functions are implemented by a hierarchical structure that consists of the metascheduler and subordinate resource managers or local batch-job management systems [8-10]. In a model, proposed in [2] there is an interaction between users launching their jobs, owners of computational resources, and VO administrators. The interests of the said users and owners are often contradictory. Each independent user is interested in the earliest launch of his job with the lowest costs (for example, the resource usage fee) and the owners, on the contrary, try to make the highest income from their resources. VO administrators are interested in maximizing the whole VO performance in the way that satisfies both users and owners [8].

In this work, economic mechanisms are applied for job batch scheduling in VO. It is supposed that resources are non-dedicated, that is along with global flows of external users' jobs, owner's local job flows exist inside the resource domains (clusters, computational nodes equipped with multicore processors, etc.). The metascheduler [8-10] implements the VO economic policy based on local system schedules. The local schedules are sets of slots coming from local resource managers or schedulers in the node domains. A single slot is a time span that can be assigned to a task, which is a part of a parallel job. We assume that job batch scheduling runs iteratively on periodically updated local schedules [2]. The launch of any job requires co-allocation of a specified number of slots. The challenge is that slots associated with different resources may have arbitrary start and finish points that do not coincide. In its turn, tasks of the parallel job must start synchronously. If the necessary number $N$ of slots with attributes matching the resource request is not accumulated then the job will not be launched. This job is joined another batch, and its scheduling is postponed till the next iteration.

We propose two algorithms for slot selection that feature linear complexity $O(m)$, here $m$ is the number of available time-slots. Existing slot search algorithms, such as backfilling [11, 12], do not support environments with heterogeneous and non-dedicated resources, and, moreover, their execution time grows substantially with increase of the number of slots. Backfilling is able to find an exact number of concurrent slots for tasks with identical resource requirements and homogeneous resources. We take a step further, so proposed algorithms deal with heterogeneous resources and jobs with different tasks.

The paper is organized as follows. Section 2 introduces a scheduling scheme. In section 3 two algorithms for search of alternative slot sets are considered. The example of slot search is presented in section 4. Simulation results for comparison of proposed algorithms are described in Section 5. Experimental results are discussed in section 6. Section 7 summarizes the paper and describes further research topics.

## 2 Scheduling Scheme

Let $J = \{j_1,..., j_n\}$ denote a batch consisting of $n$ jobs. A job $j_i, i = 1,...,n$, schedule is formed as a set $\bar{s}_i$ of time slots. A job batch schedule is a set of slot sets (*a slot combination*) $\bar{s} = (\bar{s}_1,...,\bar{s}_n)$ for jobs composing this batch. The job resource requirements are arranged into a resource request containing a wall clock time $t_i$ and characteristics of computational nodes (clock speed, RAM volume, disk space,

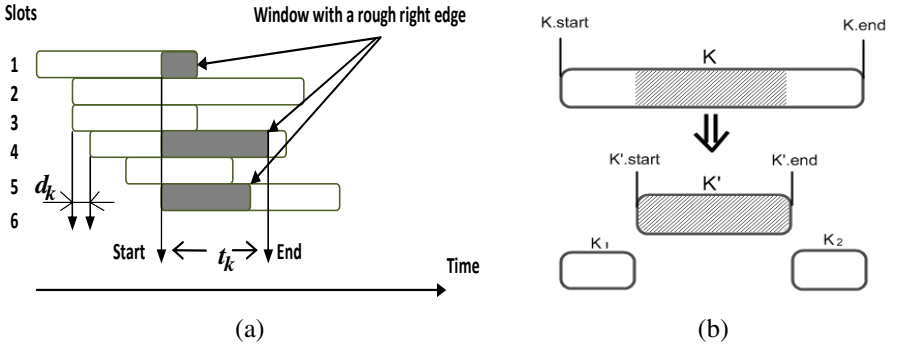
operating system etc.). The slot set $\bar{s}_i$ fits the job $j_i$, if it meets the requirements of number and type of resources, cost and the job wall time $t_i$. We suppose that for each job $j_i$ in *the current scheduling iteration* there is at least one suitable set $\bar{s}_i$. Otherwise, the scheduling of the job is postponed to *the next iteration*. Every slot set $\bar{s}_i$ for the execution of the $i$-th job in the batch $J = \{j_1,..., j_n\}$ is defined with a pair of parameters, the cost $c_i(\bar{s}_i)$ and the time $t_i(\bar{s}_i) \le t_i$ for the resource usage, $c_i(\bar{s}_i)$ denotes a total cost of slots in the set $\bar{s}_i$ and $t_i(\bar{s}_i)$ denotes a time elapsed from the start till the end of the $i$-th job. Notice that different jobs $j_{i_1}$, $j_{i_2} \in J$ have different resource requirements, and $c_{i_1}(\bar{s}) \neq c_{i_2}(\bar{s})$, $t_{i_1}(\bar{s}) \neq t_{i_2}(\bar{s})$, $i_1, i_2 \in \{1,...,n\}$, even if jobs $j_{i_1}$, $j_{i_2}$ are allocated to the same slot set $\bar{s}$. Here $c_{i_1}(\bar{s})$, $c_{i_2}(\bar{s})$ are functions of a cost $C$ of slot usage per time unit.

Two problems have to be solved for job batch scheduling. First, selecting alternative slot sets for jobs of the batch that meet the requirements (resource, time, and cost). Second, choosing the slot combination $\bar{s} = (\bar{s}_1,..., \bar{s}_n)$ that would be the efficient or optimal one in terms of the whole job batch execution.

To realize the scheduling scheme described above, first of all, we need to propose the algorithm of finding a set of alternative slot sets.

Slots are arranged by start time in non-decreasing order in a list (Fig. 1 (a)). In Fig. 1 (a), $d_k$ denotes a time offset of the slot $s_k$ in relation to the slot $s_{k-1}$.

In the case of homogeneous nodes, the set $\bar{s}_i$ of slots for the job $j_i$ is represented with a rectangle window $N \times t_i(\bar{s}_i)$. It does not mean that processes of any parallel job would finish their work simultaneously. Here a time length of the window is the time $t_i(\bar{s}_i)$ dedicated for the resource usage. In the case of nodes with varying performance, that will be a window with a rough right edge, and the resource usage time is defined by the execution time $t_k$ of the task that is using the slowest node (see Fig. 1 (a)).




(a) (b)

**Fig. 1**. Slot selection for heterogeneous resources: an ordered list of available slots (a); slot subtraction (b)

The scheduling scheme works iteratively, during *the iteration* it consecutively searches for a single alternative for each job of the batch. In case of successful slot selection for the $i$-th job, the list of vacant slots for the $(i+1)$-th job is modified. All time spans that are involved in the $i$-th job alternative are excluded from the list of vacant slots (Fig. 1 (b)). The selection of slots for the $(i+1)$-th job is performed on the list modified with the method described above. Suppose, for example, that there is a slot $\mathrm{K}'$ among the slots belonging to the same window. Then its start time equals to the start time of the window: K′.startTime = window.startTime and its end time equals to K′.end=K′.start $+t_{k'}$, where $t_{k'}$ is the evaluation of a task runtime on the appropriate resource, on which the slot $\mathrm{K}'$ is allocated. Slot K′ should be subtracted from the original list of available slots. First, we need to find slot K – the slot, part of which is K′ and then cut K′ interval from K. So, in general, we need to remove slot K′ from the ordered slot list and insert two new slots $\mathrm{K}_1$ and $\mathrm{K}_2$. Their start, end times are defined as follows: $\mathrm{K}_1$.startTime = K.startTime, $\mathrm{K}_1$.endTime = K′.startTime, $\mathrm{K}_2$.startTime = K′.endTime, $\mathrm{K}_2$.endTime = K.endTime. Slots $\mathrm{K}_1$ and $\mathrm{K}_2$ have to be added to the slot list given that the list is sorted by non-decreasing start time order (see Fig. 1 (a)). Slot $\mathrm{K}_1$ will have the same position in the list as slot K, since they have the same start time. If slots $\mathrm{K}_1$ and $\mathrm{K}_2$ have a zero time span, it is not necessary to add them to the list. After the last of the jobs is processed, the algorithm starts next search from the beginning of the batch and attempts to find other alternatives on the modified slot list. Alternatives found do not intersect in processor time, so every job could be assigned to some set of found slots without the revision of other jobs assignments. The search for alternatives ends when on the current list of slots the algorithm cannot find any suitable set of slots for any of the batch jobs. Implementation of the single alternative search algorithm becomes a serious question because characteristics of a resulting set of slots solely depend on it. Doing a search in every scheduling iteration imposes a requirement of an algorithm having complexity as low as possible. An optimization technique for choosing optimal or efficient slot combinations was proposed in [2]. It is implemented by dynamic programming methods using multiple criteria in accordance with the VO economic policy.

We consider two types of criteria in the context of our model. These are the execution cost and time measures for the job batch $J$ using the suitable slot combination $\bar{s} = (\bar{s}_1,...,\bar{s}_n)$. The first criteria group includes the total cost of the job batch execution $C(\bar{s}) = \sum_{i=1}^{n} c_i(\bar{s}_i)$. The VO administration policy and, partially, users' interests are represented with the execution time criterion for all jobs of the batch $T(\bar{s}) = \sum_{i=1}^{n} t_i(\bar{s}_i)$. In order to forbid the monopolization of some resource usage by users, a limit $B^*$ is put on the maximum value for a total usage cost of resources in the current scheduling iteration. We define $B^*$ as a budget of the VO. The total slots occupancy time $T^*$ represents owners' urge towards the balance of global (external) and local (internal) job shares. If we consider the single-criterion optimization of the

job batch execution, then every criterion $C(\bar{s})$ or $T(\bar{s})$ must be minimized with given constraints $T^*$ or $B^*$ for the interests of the particular party - the user, the owner and the VO administrator [2].

Let $g_i(\bar{s}_i)$ be the particular function, which determines the efficiency of the slot set $\bar{s}_i$ usage for the $i$-th job. In other words, $g_i(\bar{s}_i)=c_i(\bar{s}_i)$ or $g_i(\bar{s}_i)=t_i(\bar{s}_i)$. Let $f_i(Z_i)$ be the extreme value of the particular criterion using the slot combination $(\bar{s}_i,\bar{s}_{i+1},...,\bar{s}_n)$ for jobs $j_i, j_{i+1},..., j_n$, having $Z_i$ as a total occupancy time or an usage cost. Let us define an admissible time value or a slot occupancy cost as $z_i(\bar{s}_i)$. Then $z_i(\bar{s}_i)\le Z_i \le Z^*$, where $Z^*$ is the given limit. For example, if $z_i(\bar{s}_i)=t_i(\bar{s}_i)$, then $t_i(\bar{s}_i)\le T_i \le T^*$, where $T_i$ is a total slots occupancy time $i, i+1,..., n$ and $T^*$ is the constraint for values $T_i$, that is chosen with the consideration of balance between the global job flow (user-defined) and the local job flow (owner-defined). If, for example, $z_i(\bar{s}_i)=c_i(\bar{s}_i)$, then $c_i(\bar{s}_i)\le C_i \le B^*$, where $C_i$ is a total cost of the resource usage for the jobs $i, i+1,..., n$, and $B^*$ is the budget of the VO. In the scheme of backward run [2] $Z_1 = Z^*$, $z_i(\bar{s}_i)\le Z_i \le Z^*$, $Z_i = Z_{i-1} - z_{i-1}(\bar{s}_{i-1})$, having $1<i\le n$. Notice that $g_{i_1}(\bar{s})\ne g_{i_2}(\bar{s})$, $z_{i_1}(\bar{s})\ne z_{i_2}(\bar{s})$, $i_1, i_2 \in \{1,...,n\}$, even if jobs $j_{i_1}$, $j_{i_2}$ are allocated to the same slot set $\bar{s}$.

The functional equation for obtaining a conditional (given $z_i(\bar{s}_i)$) extremum of $f_i(z_i(\bar{s}_i))$ for the backward run procedure can be written as follows:

$$f_i(Z_i) = \operatorname*{extr}_{s_i}\{g_i(\bar{s}_i) + f_{i+1}(Z_i - z_i(\bar{s}_i))\},\ i=1,...,n,\ f_{n+1}(Z_{n+1})\equiv 0, \tag{1}$$

where $g_i(\bar{s}_i)$ and $f_{i+1}(Z_i - z_i(\bar{s}_i))$ are cost or time functions.

For example, a limit put on the total time of slot occupancy by tasks may be expressed as:

$$T^* = \sum_{i=1}^{n} \sum_{s_i} [t_i(\bar{s}_i)/l_i], \tag{2}$$

where $l_i$ is the number of admissible slot sets for the $i$-th job; $[\cdot]$ means the nearest to $t_i(\bar{s}_i)/l_i$ not greater integer.

The VO budget $B^*$ may be obtained by formula (1) as the maximal income for resource owners with the given constraint $T^*$ defined by (2):

$$B^* = \max_{s_i}\{c_i(\bar{s}_i) + f_{i+1}(T_i - t_i(\bar{s}_i))\}, \tag{3}$$

where $f_{i+1}(T_i - t_i(\bar{s}_i))$ is a cost function.

In the general case of the model [2], it is necessary to use a vector of criteria, for example, $<C(\bar{s}), D(\bar{s}), T(\bar{s}), I(\bar{s})>$, where $D(\bar{s}) = B^* - C(\bar{s})$, $I(\bar{s}) = T^* - T(\bar{s})$ and $T^*$, $B^*$ are defined by (2), (3).

## 3 Slot Search Algorithms

Let us consider one of the resource requests associated with any job in the batch $J$. The resource request specifies $N$ concurrent time-slots reserved for time span $t$ with resource performance rate at least $P$ and maximal resource price per time unit not higher, than $C$.

**Class Slot** is defined to describe a single slot:

```
public class Slot{
    public Resource cpu;     //resource on which the slot
                                          is allocated
     public int cash;        // usage cost per time unit
     public int start;       // start time
     public int end;         // end time
     public int length;      // time span
     …
   }
```

**Class Window** is defined to describe a single window:

```
public class Window {
     int id;                 // window id
     public int cash;        // total cost
     public int start;       // start time
     public int end;         // end time
     public int length;      // time span
     int slotsNumber;        // number of required slots
     ArrayList<Slot> slots; // window slots
           …
   }
```

Here a slot set search algorithm for a single job and resource charge per time unit is described. It is an **A**lgorithm based on **L**ocal **P**rice of slots (ALP) with *a restriction to the cost of individual slots*. Input data include available slots list, and slots being sorted by start time in ascending order (see Fig. 1(a)). The search algorithm guarantees examination of every slot of the list. If the necessary number $N$ of slots is not accumulated, then the job scheduling is postponed until the next iteration.

**1**°. Sort the slots by start time in ascending order - see Fig. 1 (a).

**2**°. From the resulting slot list the next suited slot $s_k$ is extracted and examined.

The slot $s_k$ suits, if following conditions are met:

**a)** resource performance rate $P(s_k) \geq P$;

**b)** slot length (time span) is enough (depending on the actual performance of the slot's resource) $L(s_k) \geq tP(s_k)/P$ (see the condition **a)**);

**c)** resource charge per time unit $C(s_k) \leq C$.

If conditions **a)**, **b)**, and **c)** are met, the slot $s_k$ is successfully added to the window list.

**3**°. The expiration of the slot length means that remaining slot length $L'(s_k)$, calculated like shown in **step 2**°**b**, is not enough assuming the $k$-th slot start is equal to the last added slot start: $L'(s_k) < (t - (T_{last} - T(s_k)))P(s_k)/P$, where $T(s_k)$ is the slot's start time, $T_{last}$ is the last added slot's start time. Notice, in Fig. 1 (a), $d_k = T_{last} - T(s_k)$.

Slots whose time length has expired are removed from the list.

**4**°. Go to **step 2**°, until the window has $N$ slots.

**5**°. **End** of the algorithm.

We can move only forward through the slot list. If we run out of slots before having accumulated $N$ slots, this means a failure to find the window for a job and its scheduling is postponed by the metascheduler until the next batch scheduling iteration. Otherwise, the window becomes the alternative slot set for the job. ALP is executed for every job in the batch $J = \{j_1, ..., j_n\}$. Having succeeded in the search for window for the $j_i$-th job, the slot list is modified with subtraction of formed window slots (see Fig. 1 (b)). Therefore slots of the already formed slot set are not considered in processing the next job in the batch.

In the economic model [2] a user's resource request contains the maximal resource price requirement, that is a price which a user agrees to pay for resource usage. But this approach narrows the search space and restrains the algorithm from construction of a window with more expensive slots. The difference of the next proposed algorithm is that we replace maximal price $C$ requirement by *a maximal budget of a job*. It is an **A**lgorithm based on **M**aximal job **P**rice (AMP). The maximal budget is counted as $S = CtN$, where $t$ is a time span to reserve and $N$ is the necessary number of slots. Then, as opposed to ALP, the search target is a window, formed by slots, whose total cost will not exceed the maximal budget $S$. In all other respects, AMP utilizes the same input data as ALP.

Let us denote additional variables as follows: $N_S$ – current number of slots in the window; $M_N$ – total cost of first $N$ slots.

Here we describe AMP approach for a single job.

**1**°. Find the earliest start window, formed by $N$ slots, using ALP excluding the condition **2**°**c** (see ALP description above).

**2**°. Sort window slots by their cost in ascending order.

Calculate total cost of first $N$ slots $M_N$. If $M_N \le S$, go to **4**°, so the resulting window is formed by first $N$ slots of the current window, others are returned to the source slot list. Otherwise, go to **3**°.

**3**°. Add the next suited slot to the list following to conditions **2**°**a** and **2**°**b** of ALP. Assign the new window start time and check expiration like in the **step 3**° of ALP.

If we have $N_S < N$, then repeat the current step. If $N_S \ge N$, then go to **step 2**°.

If we ran out of slots in the list, and $N_S < N$, then we have algorithm failure and no window is found for the job.

**4**°. **End** of the algorithm.

We can state three main features that distinguish the proposed algorithms. First, both algorithms consider resource performance rates. This allows forming time-slot windows with uneven right edge (we suppose that all concurrent slots for the job must start simultaneously). Second, both algorithms consider maximum price constraint which is imposed by a user. Third, both algorithms have linear complexity $O(m)$, where $m$ is the number of available time-slots: we move only forward through the list, and never return or reconsider previous assignments.

The backfill algorithm [11, 12] has quadratic complexity $O(m^2)$, assuming that every node has at least one local job scheduled. Although backfilling supports parallel jobs and is able to find a rectangular window of concurrent slots, this can be done provided that all available computational nodes have equal performance (processor clock speed), and tasks of any job have identical resource requirements.

## 4 AMP Search Example

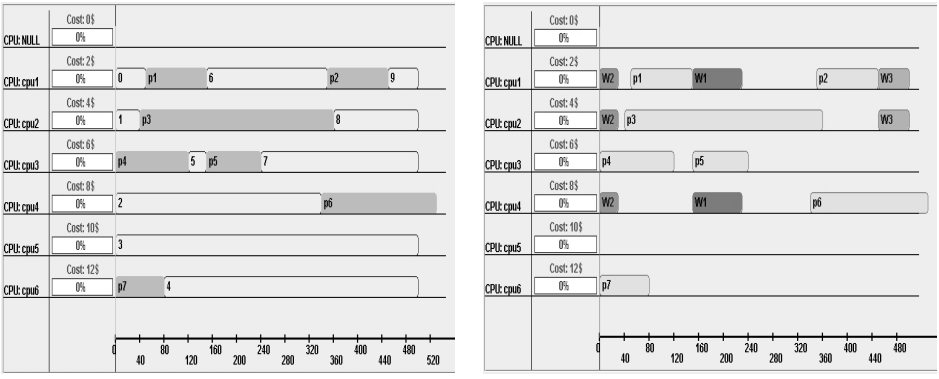
In this example for the simplicity and ease of demonstration we consider the problem with a uniform set of resources, so the windows will have a rectangular shape without the rough right edge. Let us consider the following initial state of the distributed computing environment. In this case there are six computational nodes `cpu1 - cpu6` (resource lines) (Fig. 2 (a)). Each has its own unit cost (cost of its usage per time unit). In addition there are seven local tasks `p1 - p7` already scheduled for the execution in the system under consideration. Available system slots are drawn as rectangles 0...9 - see Fig. 2 (a). Slots are sorted by non-decreasing time of start and the order number of each slot is indicated on its body. For the clarity, we consider the situation where the scheduling iteration processes the batch of only three jobs with the following resource requirements.

**Job 1** requirements:
- the number of required computational nodes: 2;
- runtime: 80;
- maximum total "window" cost per time: 10.

**Job 2** requirements:
- the number of required computational nodes: 3;
- runtime: 30;
- maximum total "window" cost per time: 30.

**Job 3** requirements:
- the number of required computational nodes: 2;
- runtime: 50;
- maximum total "window" cost per time: 6.

According to AMP alternatives search, first of all, we should form a list of available slots and find the earliest alternative (the first suitable window) for the first job of the batch. We assume that **Job 1** has the highest priority, while **Job 3** possesses the lowest priority. The alternative found for **Job 1** (see Fig. 2 (b)) has two rectangles on `cpu1` and `cpu4` resource lines on a time span [150, 230] and named W1. The total cost per time unit of this window is 10. This is the earliest possible window satisfying




(a) (b)

**Fig. 2.** AMP search example: initial state of environment (a); alternatives found after the first iteration (b)

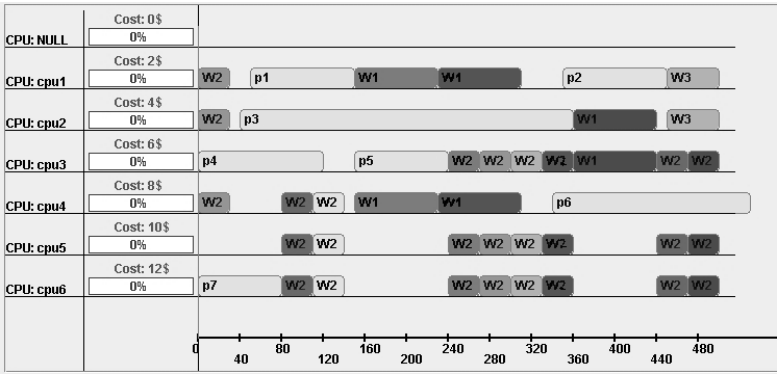



**Fig. 3.** The final chart of all alternatives found during AMP search

the job's resource request. Note that other possible windows with earlier start time are not fit the total cost constraint. Then we need to subtract this window from the list of available slots and find the earliest suitable set of slots for the second batch job on the modified list.

Further, a similar operation for the third job is performed (see Fig. 2 (b)). Alternative windows found for each job of the batch are named W1, W2, and W3 respectively. The earliest suitable window for the second job (taking into account alternative W1 for the first job) consists of three slots on the `cpu1`, `cpu2` and `cpu4` resource lines with a total cost of 14 per time unit. The earliest possible alternative for the third job is W3 window on a time span of [450, 500]. Further, taking into account the previously found alternatives, the algorithm performs the searching of next alternative sets of slots according to the job priority. The algorithm makes an attempt to find alternative windows for each batch job.

Figure 3 illustrates the final chart of all alternatives found during search.

Note that in ALP approach the restriction to the cost of individual slots would be equal to 10 for **Job 2** (as it has a restriction of total cost equals to 30 for a window allocated on three nodes). So, the computational resource `cpu6` with a 12 usage cost value is not considered during the alternative search with ALP algorithm. However it is clear that in the presented AMP approach eight alternatives have been found. They use the slots allocated on the `cpu6` resource line, and thus fit in the limit of the window total cost.

## 5 Simulation Studies

The experiment consists in comparison of job batch scheduling results using different sets of suitable slots founded with described above AMP and ALP approaches. The alternatives search is performed on the same set of available vacant system slots. The generation of an ordered list of vacant slots and a job batch is performed during the single simulated scheduling iteration. To perform a series of experiments we found it more convenient to generate the ordered list of available slots (see Fig. 1 (a)) with preassigned set of features instead of generating the whole distributed system model and obtain available slots from it.

**SlotGenerator** and **JobGenerator** classes are used to form the ordered slot list and the job batch during the experiment series. Here is the description of the input parameters and values used during the simulation. All job batch and slot list options are random variables that have a uniform distribution inside the identified intervals.

**SlotGenerator**

- number of available system slots in the ordered list varies in [120, 150];
- length of the individual slot is in [50, 300] - here we propose that the length of initial slot are varies greatly, and it will be more during the search procedure;
- computational nodes performance range is [1, 3], so that the environment is relatively homogeneous;
- the probability that the nearby slots in the list have the same start time is 0.4; this property represents that in real systems resources are often reserved and occupied in domains (clusters), so that after the release, the appropriate slots have the same start time;
- the time between neighboring slots in the list is in [0, 10], so that at each moment of time we have at least five different slots ready for utilization;
- the price of the slot is randomly selected from [0.75p, 1.25p], where p = (1.7) to the (Node Performance); here we propose that the price is a function of performance with some element of randomness.

**JobGenerator**

- number of jobs in the batch is in [3, 7]; the batch is not very big because we have to distribute all the jobs in order to carry out the experiment;
- number of computational nodes to find is in [1, 6];

- length (representing the complexity) of the job is in [50, 150]; this value is corresponds to the initial values of the generated slots;
- the minimum required nodes' performance is in [1, 2]; some jobs will require slots, allocated on resources with high ( $P \geq 2$ ) performance - it is a factor of job heterogeneity.

Let us consider the task of slot allocation during the *job batch execution time minimization*: $\min_{s_i} T(\bar{s})$ with the constraint $B*$.

The number of 25000 simulated scheduling iterations was carried out. Only those experiments were taken into account when all of the batch jobs had at least one suitable alternative of execution. AMP algorithm exceeds ALP by 35% with respect to $T(\bar{s})$. An average job execution time for alternatives found with ALP was 59.85, and for alternatives found with AMP - 39.01 (Fig. 4 (a)).

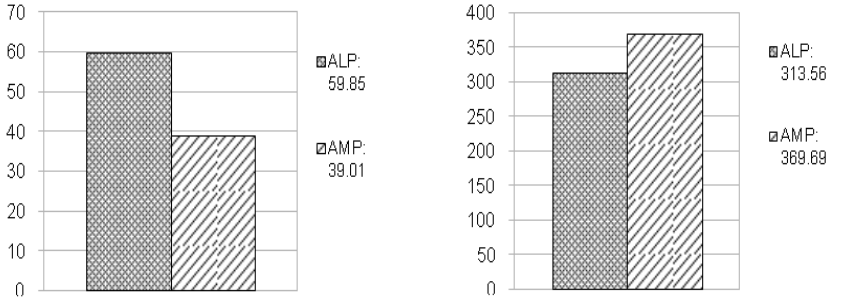



**Fig. 4.** Job batch execution time minimization: average job execution time (a); average job execution cost (b)

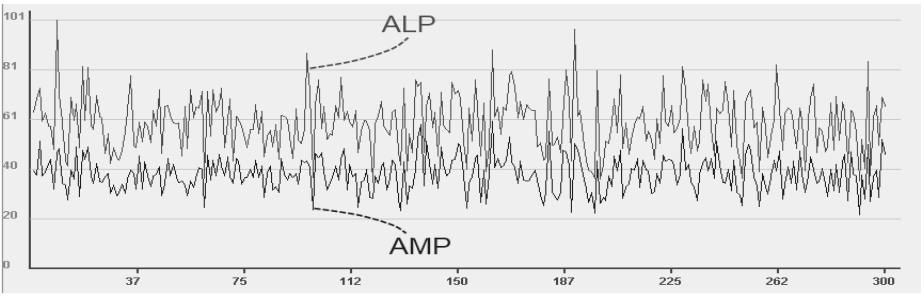



**Fig. 5.** Average job execution time comparison for ALP and AMP for the first 300 experiments in the job batch execution time minimization

It should be noted, that an average job execution cost for ALP method was 313.56, while using AMP algorithm the average job execution cost was 369.69, that is 15% more – see Fig. 4 (b).

Figure 5 illustrates scheduling results comparison for the first 300 experiments (the horizontal axis). It shows an observable gain of AMP method in every single experiment. The total number of alternatives found with ALP was 258079 or an average of 7.39 for a job. At the same time the modified approach (AMP) found 1160029 alternatives or an average of 34.28 for a single job. According to the results of the experiment we can conclude that the use of AMP minimizes the batch execution time though the cost of the execution increases. Relatively large number of alternatives found increases the variety of choosing the efficient slot combination [2] using the AMP algorithm.

Now let us consider the task of slot allocation during the *job batch execution cost minimization*: $\min_{s_i} C(\bar{s})$ with the constraint $T*$. The results of 8571 single experiments in which all jobs were successfully assigned to suitable slot combinations using both slot search procedures were collected.

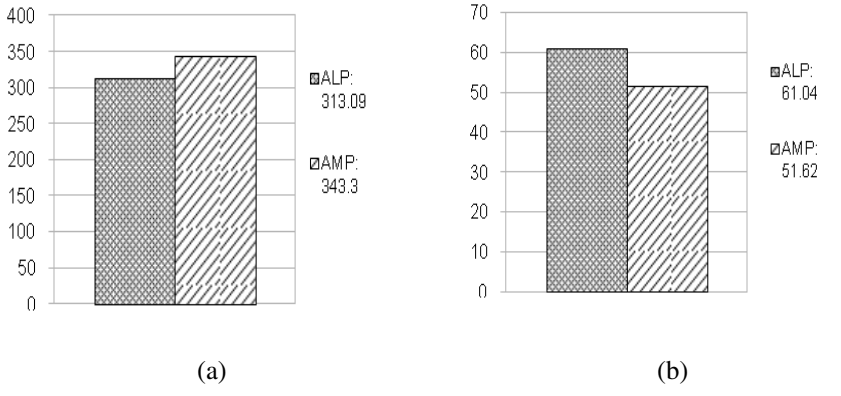



**Fig. 6.** Job batch execution cost minimization: average job execution cost (a); average job execution time (b)

The average job execution cost for ALP algorithm was 313.09, and for alternatives found with AMP - 343.3. It shows the advantage of only 9% for ALP approach over AMP (Fig. 6 (a)). The average job execution time for alternatives found with ALP was 61.04. Using AMP algorithm the average job execution time was 51.62, that is 15% less than using ALP (Fig. 6 (b)).

The average number of slots processed in a single experiment was 135.11. This number coincides with the average number of slots for all 25000 experiments, which indicates the absence of decisive influence of the available slots number to the number of successfully scheduled jobs.

The average number of jobs in a single scheduling iteration was 4.18. This value is smaller than average over all 25000 experiments. With a large number of jobs in the

batch ALP often was not able to find alternative sets of slots for certain jobs and an experiment was not taken into account.

The average number of alternatives found with ALP is 253855 or an average of 7.28 per job. AMP algorithm was able to found the number of 115116 alternatives or an average of 34.23 per job. Recall that in previous set of experiments these numbers were 7.39 and 34.28 alternatives respectively.

## 6 Experimental Results Analysis

Considering the results of the experiments it can be argued that the use of AMP approach on the stage of alternatives search gives clear advantage compared to the usage of ALP. Advantages are mostly in the large number of alternatives found and consequently in the flexibility of choosing an efficient schedule of batch execution, as well as that AMP provides the job batch execution time less than ALP.

AMP allows searching for alternatives among the relatively more expensive computational nodes with higher performance rate. Alternative sets of slots found with ALP are more homogeneous and do not differ much from each other by the values of the total execution time and cost. Therefore job batch distributions obtained by optimizations based on various criteria [2] do not differ much from each other either.

The following factors should explain the results. First, let us consider the peculiarities of calculating a slot usage total cost $C_t = CtN / P$, where $C$ is a cost of slot usage per time unit, $P$ is a relative performance rate of the computational node on which the slot is allocated, and $t$ is a time span, required by the job in assumption that the job will be executed on the etalon nodes with $P = 1$. In the proposed model, generally, the higher the cost $C$ of slot the higher the performance $P$ of the node on which this slot is allocated. Hence, the job execution time $t / P$ correspondingly less. So, the high slot cost per time unit is compensated by high performance of the resource, so it gets less time to perform the job and less time units to pay for. Thus, in some cases the total execution cost may remain the same even with the more "expensive" slots. The value $C / P$ *i*s a measure of a slot price/quality ratio. By setting in the resource request the maximum cost $C$ of an individual slot and the minimum performance rate $P$ of a node the user specifies the minimum acceptable value of price/quality. The difference between ALP and AMP approaches lies in the fact that ALP searches for alternatives with suitable price/quality coefficient among the slots with usage cost no more than $C$. AMP performs the search among all the available slots (naturally, both algorithms still have the restriction on the minimum acceptable node performance). This explains why alternatives found with AMP have on the average less execution time. Second, it should be noted that during the search ALP considers available slots regardless of the entire window. The ALP window consists of slots each of which has the cost value no more than $C$. At the same time AMP is more flexible. If at some step a slot with cost on $\delta$ cheaper than $C$ was added to the desired window, then AMP algorithm will consider to add slots with cost on the $\delta$ more expensive than $C$ on the next steps. Naturally, in this case it will take

into account the total cost restriction. That explains, why the average job execution cost is more when using the AMP algorithm, it seeks to use the entire budget to find the earliest suitable alternative.

Another remark concerns the algorithms' work on the same set of slots. It can be argued that *any* window which could be found with ALP can also be found by AMP. However, there could be windows found with AMP algorithm which can't be found with a conventional ALP. It is enough to find a window that would contain at least one slot with the cost more than $C$.

This observation once again explains the advantage of AMP approach by a number of alternatives found. The deficiency of AMP scheme is that batch execution cost on the average always higher than the execution cost of the same batch scheduled using ALP algorithm. It is a consequence of a specificity of determining the value of a budget limit and the stage of job batch scheduling [2]. However, it is possible to reduce the job batch execution cost reducing the user budget limit for every alternative found during the search, which in this experiment was limited to $S = CtN$. This formula can be modified to $S = \rho CtN$, where $\rho$ is a positive number less than one, e.g. 0.8. Variation of $\rho$ allows to obtain flexible distribution schedules on different scheduling periods, depending on the time of day, resource load level, etc.

## 7 Conclusion and Future Work

In this paper, we address the problem of independent batch jobs scheduling in heterogeneous environment with non-dedicated resources.

The scheduling of the job batch consists of two steps. First of all, the independent sets of suitable slots (alternatives of execution) have to be found for every job of the batch. The second step is selecting the efficient combination of alternative slot sets, that is the set of slot sets for the batch. The feature of the approach is searching for a number of job alternative executions and consideration of economic policy in VO and financial user requirements on the stage of a single alternative search. For this purpose ALP and AMP approaches for slot search and co-allocation were proposed and considered. According to the experimental results it can be argued that AMP allows to find on the average more rapid alternatives and to perform jobs in a less time. But the of job batch execution using AMP is relatively higher. AMP exceeds ALP significantly during the batch execution time minimization. At the same time during the execution cost minimization the gain of ALP method is negligible. It is worth noting, that on the same set of vacant slots AMP in comparison with ALP finds several time more execution alternatives.

In our future work we will address the problem of slot selection for the whole job batch at once and not for each job consecutively. Therewith it is supposed to optimize the schedule "on the fly" and not to allocate a dedicated phase during each scheduling iteration for this optimization. We will research pricing mechanisms that will take into account supply-and-demand trends for computational resources in virtual organizations.

The necessity of guaranteed job execution at the required quality of service causes taking into account the distributed environment dynamics, namely, changes in the number of jobs for servicing, volumes of computations, possible failures of

computational nodes, etc. [13]. As a consequence, in the general case, a set of versions of scheduling, or a strategy, is required instead of a single version [13, 14]. In our further work we will refine resource co-allocation algorithms in order to integrate them with scalable co-scheduling strategies.

**Acknowledgments.** This work was partially supported by the Council on Grants of the President of the Russian Federation for State Support of Leading Scientific Schools (SS-7239.2010.9), the Russian Foundation for Basic Research (grant no. 09-01-00095), the Analytical Department Target Program "The higher school scientific potential development" (projects nos. 2.1.2/6718 and 2.1.2/13283), and by the Federal Target Program "Research and scientific-pedagogical cadres of innovative Russia" (State contracts nos. P2227 and 16.740.11.0038).

## References

1. Garg, S.K., Buyya, R., Siegel, H.J.: Scheduling Parallel Applications on Utility Grids: Time and Cost Trade-off Management. In: 32nd Australasian Computer Science Conference (ACSC 2009), pp. 151–159 (2009)
2. Toporkov, V.V., Toporkova, A., Tselishchev, A., Yemelyanov, D., Bobchenkov, A.: Economic Models of Scheduling in Distributed Systems. In: Walkowiak, T., Mazurkiewicz, J., Sugier, J., Zamojski, W. (eds.) Monographs of System Dependability. Dependability of Networks, vol. 2, pp. 143–154. Oficyna Wydawnicza Politechnki Wroclawskiej, Wroclaw (2010)
3. Degabriele, J.P., Pym, D.: Economic Aspects of a Utility Computing Service. Technical Report HPL-2007-101, Trusted Systems Laboratory, HP Laboratories, Bristol (2007)
4. Pandey, S., Barker, A., Gupta, K.K., Buyya, R.: Minimizing Execution Costs when Using Globally Distributed Cloud Services. In: 24th IEEE International Conference on Advanced Information Networking and Applications, pp. 222–229. IEEE Press, New York (2010)
5. Bredin, J., Kotz, D., Rus, D.: Economic Markets as a Means of Open Mobile-Agent Systems. In: Mobile Agents in the Context of Competition and Cooperation (MAC3), pp. 43–49 (1999)
6. Buyya, R., Abramson, D., Giddy, J.: Economic Models for Resource Management and Scheduling in Grid Computing. J. of Concurrency and Computation: Practice and Experience 5(14), 1507–1542 (2002)
7. Ernemann, C., Hamscher, V., Yahyapour, R.: Economic Scheduling in Grid Computing. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 128–152. Springer, Heidelberg (2002)
8. Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J.: Multicriteria Aspects of Grid Resource Management. In: Nabrzyski, J., Schopf, J.M., Weglarz, J. (eds.) Grid resource management. State of the art and future trends, pp. 271–293. Kluwer Academic Publishers, Dordrecht (2003)
9. Toporkov, V.: Application-Level and Job-Flow Scheduling: An Approach for Achieving Quality of Service in Distributed Computing. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 350–359. Springer, Heidelberg (2009)
10. Toporkov, V.V.: Job and Application-Level Scheduling in Distributed Computing. Ubiquitous Computing and Communication J. 3(4), 559–570 (2009)

11. Mu'alem, A.W., Feitelson, D.G.: Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. IEEE Transactions on Parallel and Distributed Systems 6(12), 529–543 (2001)
12. Jackson, D.B., Snell, Q.O., Clement, M.J.: Core Algorithms of the Maui Scheduler. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 87–102. Springer, Heidelberg (2001)
13. Toporkov, V.V., Tselishchev, A.: Safety Scheduling Strategies in Distributed Computing. International Journal of Critical Computer-Based Systems 1/2/3 (1), 41–58 (2010)
14. Toporkov, V.V., Toporkova, A., Tselishchev, A., Yemelyanov, D.: Scalable Co-Scheduling Strategies in Distributed Computing. In: 5th ACS/IEEE Int. Conference on Computer Systems and Applications, pp. 1–8. IEEE CS Press, New York (2010)

# An Initial Approximation to the Resource-Optimal Checkpoint Interval

Ekaterina Tyutlyaeva[1] and Alexander Moskovsky[2]

[1] Program System Institute of RAS, Pereslavl-Zalessky, Yaroslavl Region, 152020, Russia
ordi@xgl.pereslavl.ru
[2] ZAO "RSK SKIF", Moscow, Russia

**Abstract.** In this paper, some basic considerations are made about cost-effective checkpointing in a cloud-like environment. We deduce a cost model of a task considering expected costs of computational and storage resources and time-to-solution costs. With the simple analytic expression, we calculate efficient checkpoint intervals relying on cost parameters, taken from real-world computational clouds. Results demonstrate reasonability of the model proposed and we discuss its potential implications in a high-performance computing environment.

**Keywords:** Fault-tolerance, optimal checkpointing, resource cost, cloud computing.

## 1 Introduction

The increasing scalability of modern computational clusters and the need to solve larger and more complex problems make reliability crucial for HPC development. It is significant to note that reliability support implies considerable overhead costs. These costs include additional evaluation time, additional CPU, RAM, sometimes I/O operations.

Simultaneously, high-performance compute clouds have emerged as a new pay-as-you-go model with access to various resources (latest offerings of Amazon [1]).

In this work we have tried to investigate how to set optimum checkpoint interval simply with respect to overhead time, CPU, RAM and I/O costs, considering the simplest example of a single process running on a computer system of limited reliability. If we see that a simple model gives reasonable figures, we could consider applying it to more complex cases, including more sophisticated fault-resilience schema based on dataflow.

## 2 Model Formulation

The major computing system resources are:

- CPU-time
- RAM memory

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 384–389, 2011.

– Storage system (disk memory)
– Time (waiting for a result)

Each resource has a cost associated with it. The goal is to minimize overall solution costs.

In the current consideration, a studied computational process comprises several stages with approximately even resource workload. For example, the first stage is most likely related to I/O workload, the computational stage is attributed to CPU and RAM workload. In this model we will consider the uniform process stage with an even workload and under the hypothesis that checkpointing resource costs and time overheads are similar in every moment of this stage.

## 3 Related Work

There is a relatively large body of work that has goal to optimize overall time of program execution. Young [2] proposed useful approximation of the optimum checkpoint interval for restart dumps. Daly [3] proposed a higher order estimation of the optimum checkpoint interval taking into account the amount of time required to restart and varying fraction of a segment required to rework and improve segment size to failure. Another work, "Stochastic Models for Restart, Rejuvenation and Checkpointing" [4] includes stochastic models for checkpointing with respect to maximum system availability, minimum task completion time or minimal total recovery and checkpointing time costs. The purpose of these research works typically is to reduce the job completion time by optimizing the time between checkpoints.

Another related work, "Cost-Based Oracle Fundamentals" [5] estimates query cost with respect to statistics of data distribution and hardware performance. The Oracle query cost depends on time reading single block, multi-block and estimated CPU-time. Oracle uses cost concept for providing backward compatibility and optimizing query costs.

In our work, we have tried to make the first-order simple approximation of the optimum checkpoint interval, similar to the first Young approximation, with respect to all resources (including not only time, but CPU, RAM, disk operations "unit weight"). The purpose of the offered checkpointing is to minimize the computation resource cost by minimizing checkpointing and recovery overheads.

## 4 First Order Approximation

Generally, the considered function will look like

$$Total\_cost = solution\_cost + recomputation\_cost + checkpointing\_cost \quad (1)$$

where solution cost is the amount of resources spent on performing actual computation that represents progress towards a solution. Re-computation cost is the amount of resources spent on rollback recovery from the nearest checkpoint. Checkpoint costs are the amount of indispensable resources to create total

computational process dump. Total cost is the expectation value of overall computing time.

For determining the optimal checkpoint interval we will use the following application-related values:

$\tau$ — optimizable checkpoint interval between dumps;

$T_s$ — solution time (Solution time is defined as time spent on actual computational cycles towards a final solution). $T_s = N \cdot \tau$, where N is the number of intervals;

$\delta$ — time for saving checkpoint data;

$\mu$ — solution resources, except time. Similarly to solution time, solution resources are defined as the amount of resources required for actual computational cycles toward a final solution;

$\gamma$ — amount of resources, except time, required for saving checkpoint data;

$M$ — mean time to interrupt for the system.

Furthermore, we will define the following functions:

$\phi(\tau)$ — approximation of the fraction of a segment requiring rework (in case of a failure);

$n(\tau)$ — approximation of the expected number of failures at the interval.

Thus, using these notations, we can define the number of checkpoints as

$$\frac{T_s}{\tau} - 1 \tag{2}$$

because there is no checkpoint on the last segment.

Total rework time will be the product of expected failure number, expected failure segment and time to recount selected interval and checkpoint.

$$[\tau + \delta] \cdot \phi(\tau + \delta) \cdot n(\tau) \tag{3}$$

Finally, total solution time can be, by analogy with Young solution, expressed through the formula:

$$T_s + (\frac{T_s}{\tau} - 1)\delta + [\tau + \delta] \cdot \phi(\tau + \delta) \cdot n(\tau) \tag{4}$$

Furthermore, we'll try to express all other resources costs, except time. Useful solution resources are defined as $\mu$ and resource costs for checkpoint saving is defined as $\gamma$.

Total amount of resources (except time):

$$\mu + (\frac{T_s}{\tau} - 1)\gamma + [\frac{\mu\tau}{T_s} + \gamma] \cdot \phi(\tau + \delta) \cdot n(\tau) \tag{5}$$

Assuming time is a standalone resource with weight coefficient k. Then, total amount of resources with selected dump interval will be (see Eq. 4 and Eq. 5)

$$R(\tau) = k(T_s + (\frac{T_s}{\tau} - 1)\delta + [\tau + \delta] \cdot \phi(\tau + \delta) \cdot n(\tau)) + \tag{6}$$

$$+\mu + (\frac{T_s}{\tau} - 1)\gamma + [\frac{\mu\tau}{T_s} + \gamma] \cdot \phi(\tau + \delta) \cdot n(\tau) \tag{7}$$

According to Young [2] and Daly [3] propositions, we will use halfway through the compute interval as the fraction of rework in our first-order model:

$$\phi(\tau) = \frac{1}{2} \tag{8}$$

Assuming that interrupt distribution is similar to the Poisson distribution, we will approximate expected number of failures on interval $\tau$ by analogy with Young:

$$n(\tau) = T_s(e^{\frac{\tau+\delta}{M}} - 1) \simeq \frac{T_s}{\tau} \cdot (\frac{\tau+\delta}{M}), for \frac{\tau+\delta}{M} \ll 1 \tag{9}$$

Let's substitute $\phi(\tau)$ 8 and $n(\tau)$ into the main estimating formula 6:

$$R(\tau) = k(T_s + (\frac{T_s}{\tau} - 1)\delta + [\tau+\delta] \cdot \frac{1}{2} \cdot \frac{T_s}{\tau} \cdot (\frac{\tau+\delta}{M})) + \tag{10}$$

$$+\mu + (\frac{T_s}{\tau} - 1)\gamma + [\frac{\mu\tau}{T_s} + \gamma] \cdot \frac{1}{2} \cdot \frac{T_s}{\tau} \cdot (\frac{\tau+\delta}{M}) \tag{11}$$

In order to search a single optimum value $\tau$, we will consider the first derivative of evaluation function 10 and assuming that the $\delta$ squared term is negligible (according to 9) , we receive a unique positive solution:

$$\tau = \sqrt{(\frac{2kT_sM\delta + 2MT_s\gamma + T_s\delta\gamma}{\mu + kT_s})} \tag{12}$$

Using $k = 0$ we can evaluate an optimal time interval for resource cost minimization nonregistering time. Contrariwise, nulling weight coefficient of other resources $\mu, \gamma$ allows us to recover Young's original solution and evaluate time-efficient dump interval.

## 5 Calculation of Optimal Interval by Examples

Now let's substitute some figures into the formula, which seems reasonable. The cost per point of each computational resource can be approximated according to Amazon resource prices - Amazon EC2 [1]. As an example, let's examine prices for an extra-large problem, which are applicable for CPU-intensive instances. CPU cost for Extra large instance is $0.68 per hour, memory $0.50 and I/O requests $0.10 per one million. The resource cost will be determined as weighted sum of CPU, RAM and I/O load, where weight coefficients will be related to the Amazon EC2 service prices. In a similar manner, we can define $\mu$ and $\gamma$. Therefore with Amazon-weighted resources optimal time interval 12 will look like:

$$\sqrt{(\frac{2kT_sM\delta + 2MT_s(0.68b_1 + 0.50b_2 + 0.10b_3) + T_s\delta(0.68b_1 + 0.50b_2 + 0.10b_3)}{(0.68a_1 + 0.50a_2 + 0.10a_3) + kT_s})} \tag{13}$$

Cost of time can be considered something artificial; however, we propose to take the National Minimum Wage in the USA as a baseline. Even if a higher paid specialist is wating for the computation to be finished, we believe to have a correct order of magnitude. According to [6], Federal USA minimum wage level is $7.25 (per hour). For example, let's take an average value about 30,000 hours or 3.4 years. Let's substitute M=30000, k=7,25 and $a_1 = 350$, $a_2 = 300$, $a_3 = 150$ and $b_1 = 3$, $b_2 = 2$, $b_3 = 4$ (consider checkpointing as dumping with saving to disk, with less RAM consumption and increased time and disk operations number), then we get an optimal time interval at the Eq. 13 $\tau = 190.9$ hours. For comparison, let's substitute $M = 30000$, $T = 504$ and $\delta = \frac{1}{5}$ in the original Young approximation [2], and we'll get $\tau = 109.5$ hours.

This example demonstrates that time-optimal and resource-optimal time interval can differ significantly.

## 6 Model Limitations

It's important to underline that evaluated formula 12 defines the optimal interval only in the first approximation with a large set of limitations. First of all, our formula doesn't take into consideration such important facts as variability of checkpoint costs, ambiguity of computation cost and dependency between process stage and checkpoint costs.

## 7 Conclusions and Future Work

This work introduces a new approach to the optimal checkpointing interval research. We have proposed an initial approximation of the optimum checkpoint interval with respect to all computational resource, including CPU, RAM, disk operations and execution time. This work also requests for comment the idea of resource weight specification according to Cloud services resource prices. Furthermore, we found the first approximation to an optimal solution of resource efficient challenge:

$$\tau = \sqrt{(\frac{2kT_sM\delta + 2MT_s\gamma + T_s\delta\gamma}{\mu + kT_s})} \tag{14}$$

Proposed solution is a the natural extension of the Young time-optimal solution and allows to receive the approximate time interval for cost-optimal and time-optimal checkpointing strategy. As a first direction of the future work, we are planning to consider a higher-order approximation of the resource optimal checkpoint interval. Furthermore, we want to explore the impact of the resource-optimal checkpoint interval on real complex problems.

Recently, fault-oblivious [7] implementations of programming models are under the subject of research. The proposed cost model, while being simple, could be the guide for decision-making to enable optimal fault-resiliency mechanism in

such frameworks. In aforementioned works, fault-tolerance is enabled by coarse-grain dataflow approach, which allows re-computation of tasks, assigned to processors which failed during computation. We are planning to save those parallel tasks and intermediate results whose mathematical expectation is founded in environment of evaluated resource-optimal time-interval for each node and each computational branch separately. We will consider parallel task by analogy with database transaction within the limits of the "all-or-nothing" proposition and "pure" function concept. We expect that this approach will allows us to decrease overheads, avoid synchronization problems and implement problems with complicated execution graph. Evidently, each task has an individual CPU/memory footprint and we are planning to use reductive testing executions and special tools for observation hardware events and loading.

## References

1. Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/
2. Young, J.W.: A first order approximation to the optimum checkpoint interval. Commun. ACM 17, 530–531 (1974)
3. Daly, J.T.: A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. In: Future Generation Computer Systems, pp. 303–312 (2006)
4. Wolter, K.: Stochastic Models for Fault Tolerance: Restart, Rejuvenation and Checkpointing. Springer, Heidelberg (June 8, 2010)
5. Lewis, J.: Cost-Based Oracle Fundamentals (Expert's Voice in Oracle). Apress, USA (2005)
6. Current Minimum Wage, Federal & State Minimum Wage Rates, Laws, and Resources (2011), http://www.minimum-wage.org/history.asp
7. Abramov, S.M., Esin, G.I., Matveev, G.A., Roganov, V.A., Zagorovsky, I.M.: Principles of organization of the failure tolerance parallel computations for the computational and control tasks in the T-system with the open architecture (OpenTS). In: PSTA 2006, Pereslavl-Zalessky (2006) (in Russian)

# Performance Characteristics of Global High-Resolution Ocean (MPIOM) and Atmosphere (ECHAM6) Models on Large-Scale Multicore Cluster

Panagiotis Adamidis, Irina Fast, and Thomas Ludwig

German Climate Computing Centre (DKRZ)
Bundesstr. 45a, D-20146 Hamburg, Germany
adamidis@dkrz.de
http://www.dkrz.de

**Abstract.** Providing reliable estimates of possible anthropogenic climate change is the subject of considerable scientific effort in the climate modeling community. Climate model simulations are computationally very intensive and the necessary computing capabilities can be provided by supercomputers only. Although modern high performance computer platforms can deliver a peak performance in the Petaflop/s range, most of the existing Earth System Models (ESMs) are unable to exploit this power. The main bottlenecks are the single core code performance, the communication overhead, non-parallel code sections, in particular serial I/O, and the static and dynamic load imbalance between model partitions. The pure scalability of ESMs on massively parallel systems has become a major problem in recent years. In this study we present results from the performance and scalability analysis of the high-resolution ocean model MPIOM and the atmosphere model ECHAM6 on the large-scale multicore cluster "Blizzard" located at the German Climate Computing Center (DKRZ). The issues outlined here are common to many currently existing ESMs running on massively parallel computer platforms with distributed memory.

**Keywords:** parallel computing, performance analysis, earth system models

## 1 Introduction

The current TOP500[1] list of supercomputer systems is clearly dominated by large scale multicore clusters. In fact, only such systems can provide the computing capabilities necessary to run scientific grand challenge applications from high energy and nuclear physics, chemistry, material, life, and environmental sciences. In the last decades the relevance of the computationally and data intensive climate simulations has steadily increased, due to their significant impact

[1] http://www.top500.org/list/2010/11/100

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 390–403, 2011.

on the decisions that will affect political, economical, and social aspects of human societies. For this purpose a number of coupled Earth-System-Models (ESMs) consisting of components that describe physical and biogeochemical processes in the different parts of the Earth System (atmosphere, ocean, land, cryosphere, biosphere etc.) have been developed worldwide.

Unfortunately, performance analyses show that many of the current ESMs are not able to exploit massively parallel systems efficiently. The gap between machine peak performance and climate application sustained performance as well as limited scalability of climate models become more and more evident. The major bottlenecks identified so far are the static and dynamic load imbalance, the serial I/O, communication overhead, and use of inefficient mathematical/coupling algorithms.

In the work presented here, we focus on the performance analysis of the ocean (MPIOM) and atmosphere (ECHAM6) model components of the Earth System Model MPI-ESM. Short model and experiment descriptions are given in the next section. Thereafter, key performance characteristic of the models on the IBM p575 "Power6" cluster at the German Climate Computing Center (DKRZ) are discussed. Code optimizations of the ocean model MPIOM and corresponding performance achievements are described in the subsequent section. The paper closes with summary and outlook.

## 2 Model Description

In our work we consider the ocean (MPIOM) and atmosphere (ECHAM6) models, which are components of the global Earth System Model MPI-ESM developed at the Max-Planck-Institute for Meteorology in Hamburg, Germany.

ECHAM6 is the 6th generation Atmospheric General Circulation Model (AGCM), that is originally derived from the European Centre for Medium-Range Weather Forecasts (ECMWF) model. The model dynamics are based on approximated hydrostatic primitive equations. The model prognostic variables are vorticity, divergence, temperature, logarithm of surface pressure, water vapor, cloud liquid water, and cloud ice. The horizontal discretization is realized using spectral decomposition in spherical harmonic basis functions with transformation to a Gaussian grid for calculation of non-linear equation terms and some physical parametrizations. Hybrid sigma-pressure coordinates are used for the vertical representation. For the time-integration semi-implicit leapfrog scheme with Asselin time filter is employed. More detailed description of the model's dynamical core and all physical parametrizations (convection, clouds, boundary layer, short-wave and long-wave radiation, gravity wave drag etc.) can be found in [1].

MPIOM is an Ocean General Circulation Model (OGCM) based on the hydrostatic and Boussinesq approximated ocean primitive equations with representation of thermodynamic processes. The prognostic variables and tracers are 3D-velocity field (u, v, w), temperature, salinity, and surface elevation. MPIOM uses spherical curvilinear bipolar or tripolar coordinate system. The poles are

located on land masses to avoid grid singularity at the North Pole. For the horizontal discretization an Arakawa C-grid is used. A semi-implicit iterative solver is implemented for the barotropic part of the model. For vertical representation z-coordinates with partial bottom cells are used. MPIOM also contains an embedded dynamic/thermodynamic sea-ice model with ice thickness, ice concentration, ice velocities (u,v), and snow depth as prognostic variables. For the detailed description of MPIOM as well as of the coupled MPI-ESM model see [2] and [3].

The development of MPIOM and ECHAM models started in the late 1980's. The initial versions were serial programs written in FORTRAN77. The MPI-parallel versions were developed around 1999–2000. The parallelization was accompanied by a change from FORTRAN77 to FORTRAN95. Both models are parallelized using uniform block domain decomposition. MPIOM is a pure MPI application. ECHAM6 is hybrid parallelized with MPI and OpenMP, however the performance gain due to OpenMP threading will be not discussed in this work.

The high demands of MPIOM and ECHAM6 models on computing resources can be demonstrated on the basis of the experiments for the Assessment Reports of the Intergovernmental Panel on Climate Change (IPCC AR). In case of the last published IPCC AR4 [4] the total number of the simulated years amounted to 5000. About $4 \times 10^5$ CPU hours on a NEC SX6 parallel vector processing machine were necessary to perform the experiments. For the phase five of the Coupled Model Intercomparison Project (CMIP5), which will provide the data basis for the next IPCC AR5 (scheduled to be published in 2013), about thirty different experiments covering such topics as paleo- and historical climate simulations, near-term decadal and centennial future climate change projections, and experiments with idealized forcing will be performed. The total number of simulated years will amount to 7500. The expected computing time is about $12 \times 10^6$ CPU hours on the DKRZ IBM cluster. However, the spatial resolution for CMIP5/IPCC AR5 experiments is still insufficient to resolve many regional phenomena such as hurricanes, intense mid-latitude cyclones, ocean coastal dynamics, and topographically constrained ocean circulation features. For this reason, some selected climate change simulations of the IPCC AR5 type with the MPI-ESM at a significantly higher horizontal and vertical resolution will be performed in the framework of the project STORM[2].

For the performance analysis presented in this work we ran a number of experiments using high-resolution STORM model setups. The configurations are summarized in the Table 1. The simulated time period is one month (January).

## 3 Performance Analysis

The performance measurements were carried out on the IBM p575 "Power6" cluster "Blizzard" located at the DKRZ. The system consists of 264 compute

[2] `https://verc.enes.org/community/projects/national-projects/german-projects/storm`

**Table 1.** MPIOM and ECHAM6 configurations used in the performance analysis. The resolution acronyms TP04 or TP6M stands for Tri-Polar grid following by the mean grid point distance in geographical coordinates (MPIOM); T255 specifies the level of the triangular truncation in the wave number space of spherical harmonics (ECHAM6); L specifies the number of vertical levels.

| Model | resolution acronym | Number of grid points | Number of vertical levels | Horizontal grid spacing | Vertical grid spacing | Time step |
|---|---|---|---|---|---|---|
| MPIOM | TP04L80 | 802x404 | 80 | 0.4° | 10–280m | 3600s |
| MPIOM | TP6ML80 | 3602x2394 | 80 | 0.1° | 10–280m | 600s |
| ECHAM6 | T255L199 | 768x384 | 199 | 0.5° | 14–2581m | 90s |

nodes and is the world's largest IBM Power6 installation in a single infiniband cluster. Each node has 16 dual core 4.7GHz processors. The total number of cores is 8448. In simultaneous multi-threading (SMT) mode two separate threads can be executed concurrently on each physical core. The threads are treated as independent logical processors by the operating system and can be used to allocate MPI processes only or hybrid combinations of MPI processes and OpenMP threads. The cluster has total peak performance of 158 Teraflop/s and is currently placed on rank 58 in the TOP500 list (November 2010). "Blizzard" provides the production environment for most of the climate model simulations performed in Germany. Only due to the installation of this cluster, the high-resolution climate simulations described in this work became possible.

The parallelization algorithm of MPIOM and ECHAM6 (in grid point space) is based on domain decomposition. Thereby, the original computational domain is decomposed into rectilinear subdomains which are spread among the available processors. In Fig. 1 a 2x2 decomposition is shown. The parallel programming model used is SPMD (Single Program Multiple Data). In either model no decomposition is performed within vertical column.

According to the numerical scheme in MPIOM, each MPI process is carrying out several local computations on the subdomain it has been assigned to. These calculations can take place asynchronously and have references to variables stored in local memory. Within each time step a considerable amount of communication is needed, in order to exchange the necessary data at the boundaries of each subdomain. At this point a synchronisation takes place, since every process has to communicate the needed data with all of its neighbours, two in the east-west direction and two in the north-south direction (see Fig. 1). After this step has been accomplished, and all the variables have been updated, the program can continue with computations on the local subdomains. The execution can be described with the BSP (Bulk Synchronous Parallel) model.

In ECHAM6 forward and backward transpositions between legendre space, fourier space and grid point space are needed. The transpositions are accompanied by the redistribution of all required data, which is partially realized by means of expensive collective MPI "gather/scatter" and "alltoall" communication within defined process groups. Between two successive transpositions, interprocessor communication is hardly necessary.

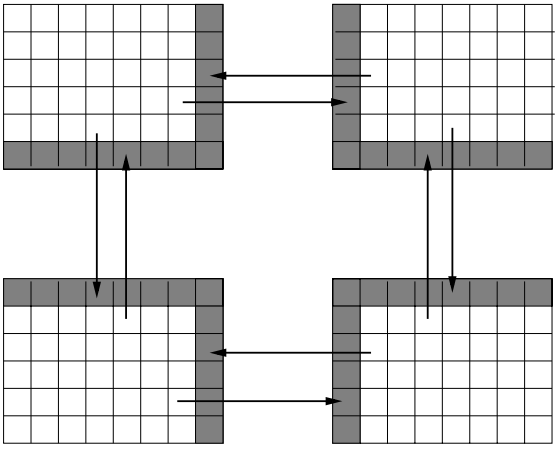


**Fig. 1.** 2x2 decomposed computational domain. Halos are gray shaded. For the sake of convenience physical boundaries are omitted here.

The I/O in MPIOM and ECHAM6 is realized in a serial manner. Formats commonly used for climate and weather data are NetCDF3/4 [7], HDF5 [8], and GRIB1/2 [6]. Only the first two standards are supported by libraries, which optionally make use of MPI-IO. A library which combines GRIB with MPI-IO does not exist yet. Furthermore, the data layout used by models is not identical to that of a file. As a consequence a major bottleneck appears due to a dynamic mapping of the application data model onto the data model of the underlying file.

In Fig. 2, the speedup curves for the ocean model MPIOM and the atmosphere model ECHAM6 are shown. Constrained by the memory requirements, the minimum number of cores on which the high resolution models can run is 512. So, unlike the conventional speedup metric definition, the presented speedup calculations refer not to the serial execution time but to the execution time on the minimal number of cores the application can be run on. The measurements cover the processes range from 512 to 4096 and imply an ideal speedup of factor 8. The ocean model MPIOM is memory bound and does not benefit from SMT mode. ECHAM6 experiments were performed in SMT mode with two MPI tasks per physical core.

In case of MPIOM, doubling the number of MPI tasks from 512 to 1024 speeds up the model by a factor of 1.86. This corresponds to an efficiency of 93%. A quadrupling of the number of MPI tasks to 2048 results in a speedup of 2.95 and an efficiency of 74%. The speedup curve stagnates for processes number larger than 2048. Finally, a clear drop of MPIOM speedup is evident for process numbers higher than 3072. In case of ECHAM6, a twofold number of processes (1024) results in a speedup of 1.81, whereas a fourfold number of processes (2048) leads to a speedup of 2.8 out of 4. An increasing degradation of the speedup curve above 1536 cores is clearly visible. For eightfold number of processes (4096) the speedup amounts to 3.7 compared to ideal speedup of 8. This corresponds to an

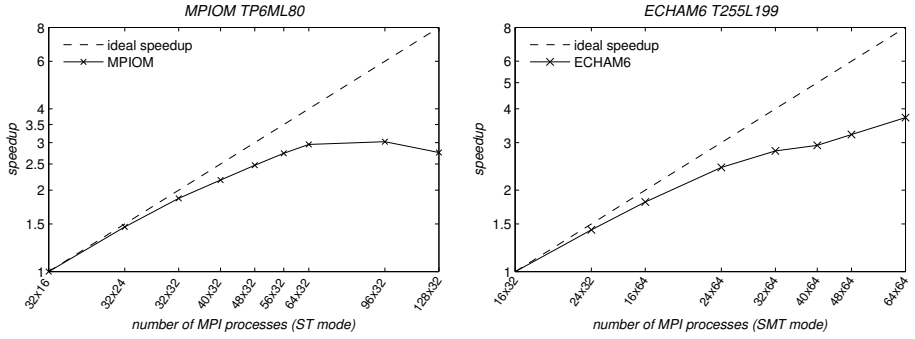



**Fig. 2.** Speedup curves for ocean MPIOM and atmosphere ECHAM6 models. Both axes are logarithmically scaled. The abscissa shows the decomposition of the computational domain in longitudinal (x) and latitudinal (y) direction, the product of both numbers corresponds to the total number of MPI processes.

efficiency of 46%. Across the performed measurements, the wallclock time needed to simulate one month changes from 29000 to 10000 seconds for MPIOM and from 41000 to 11000 seconds for ECHAM6. The observed scalability behaviour is mainly caused by the lack of the scalability of global communication as well as by increased work load imbalance through serial code sections. Furthermore, quite small partitions have the effect, that local work load variations within one partition do not compensate each other.

The missing scalability of both models has severe implications for throughput (number of simulated years per computational day) of high-resolution climate simulations. Throughput rates in the range of 20-30 simulation years per day are required to spin up and tune climate models as well as to perform ensemble simulations within reasonable time frames. At about 11000 seconds wallclock time per simulated month (for MPIOM and ECHAM6), a high-resolution climate simulation would advance by about 7-8 simulation months per day, which is far below acceptable progress. A 200-year long climate simulation running non-stop on the "Blizzard" cluster would take about 1 year.

For an in-depth MPI profiling of both models the VampirTrace tool [9] has been used. It records all calls to the MPI library and writes the collected data to a trace file. Using data from the trace file, different metrics can be derived. The relative distribution of the user and MPI time for different number of MPI processes is shown in Fig. 3. The fraction of the MPI time on the total program execution time rises dramatically with the increasing number of MPI processes. For the smallest possible configuration with 512 MPI processes it amounts to about one third. For large process numbers the application execution time is clearly dominated by the MPI communication (up to 82% or 65% for MPIOM and ECHAM6 respectively). Considerable fractions of the MPI_WAIT[ALL] time (amounting to 20-30%) hint at strong work load imbalance between different partitions.

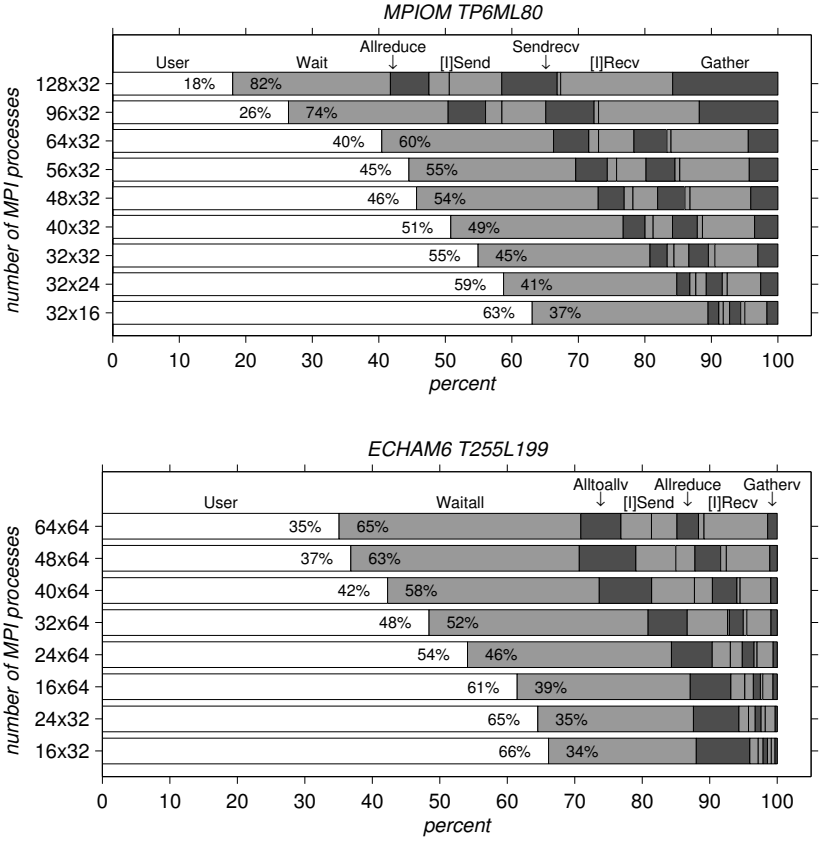



**Fig. 3.** Percentage of user (without MPI library calls) time (white) and MPI time (gray) on total execution time for different number of processes for ocean model MPIOM (top) and atmosphere model ECHAM6 (bottom). The MPI time is further subdivided into several MPI operations which are of relevance for the particular model. Mean values over all processes are depicted.

The problem of work load imbalance is illustrated in Fig. 4. For MPIOM a clear difference between land ("less work") and ocean ("more work") dominated partitions is recognizable. Also, the influence of the ocean topography is visible. For the ECHAM6 model the pattern is a bit more complicated because of the superposition of static and dynamic work load imbalances. The static part is caused by the fact that treatment of the land surface and soil is more time-consuming in comparison to the treatment of the ocean and sea ice surface. The most important contribution to the dynamic load imbalance comes from very expensive radiation calculations. To account for the seasonal and spatial patterns of the incoming solar radiation (e.g. Polar Night in the Norther Hemisphere (NH) vs. Polar Day in the Southern Hemisphere (SH)), each process in ECHAM6 executes calculations for one tile from the NH and a corresponding tile from the

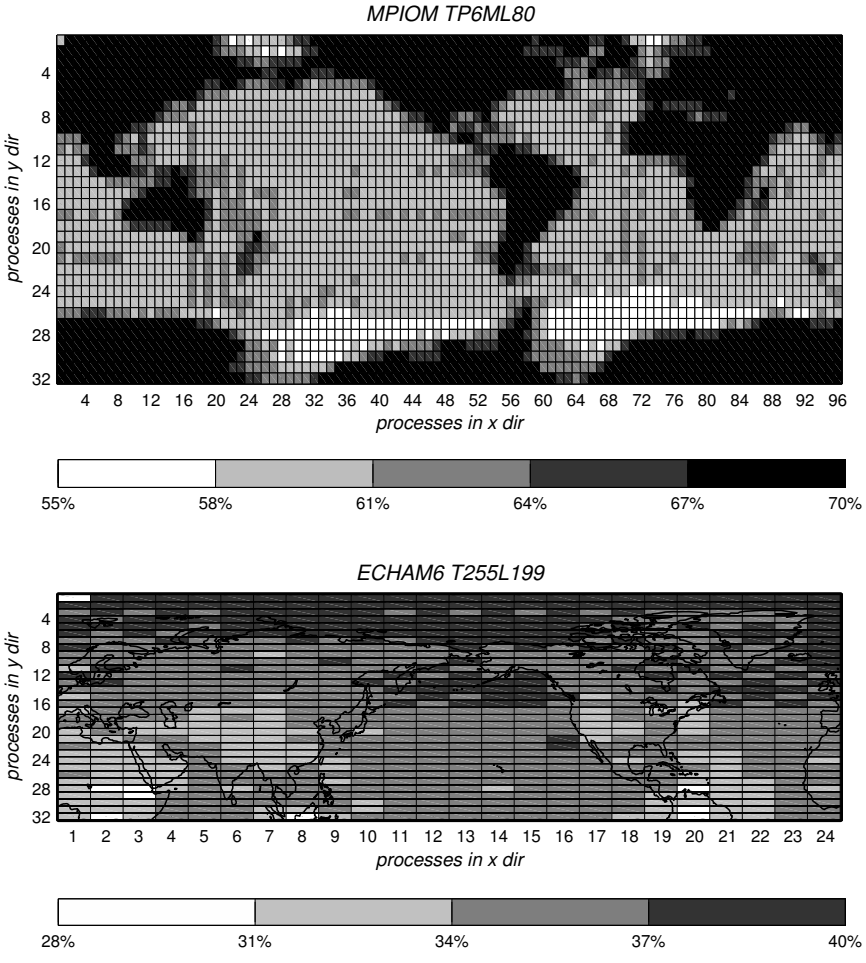



**Fig. 4.** Distribution of MPI time fraction (%) with respect to the total execution time for ocean model MPIOM with decomposition 96x32 (top) and atmosphere model ECHAM6 with decomposition 24x32 (bottom). For ECHAM6 the distribution is axially symmetric about the equator. Partitions with lower work load reveal longer MPI times due to waiting for processes with higher work load.

SH. Obviously, this approach can bring only partial compensation of the "solar altitude" effect.

In both models the master process (partition[1,1] in Fig. 4) is clearly recognizable. On this process global fields are gathered for calculation of some global diagnostics and for serial output of the model data. With the increasing number of processes the differentiation between the heavily work loaded master process and other processes is becoming more and more evident (not shown). This counteracts the scaling of the models.

Another worrying issue is the flop rate achieved. For MPIOM, the mean percentage of peak performance is about 3%, which is far too low, especially for future exascale supercomputers, consisting of several tens or hundreds of thousands of cores. One of the main reasons for this is that the ocean model MPIOM is memory bound and much CPU time is spent to move data between memory and registers.

## 4 MPIOM Code Optimizations

Due to the high complexity of the climate models, fully productive codes, which are optimally adapted to the massively parallel systems, will hardly be available for the next couple of years. To bridge the transition from the legacy code climate models to radically revised or newly developed climate models, it is important to find practicable solutions aimed at improvement of the performance of existing codes. Here we describe some results from the optimizations of the ocean model MPIOM, that mainly consist of the adaptations of the computationally intensive model routines to the IBM p575 architecture.

### 4.1 Improving Computation

A very crucial issue is single core performance. Applying the principle of locality [5] on climate models has proven to be a difficult task. Techniques like cache blocking or loop transformations are quite successful when used for linear algebra kernels, like those found in the LINPACK benchmark suite, but their implementation in climate models is not straightforward.

The complexity of the numerical schemes used in climate models is much higher than in basic linear algebra operations, which means that major algorithmic changes are necessary in order to write a code which performs well on cache based architectures. The outcome of the above mentioned changes, does not always deliver bit identical results thus influencing the results of the simulations. In other words, the frequency of reevaluating of model's physics puts major constraints in the pace at which the code can be optimized.

The Power6 microprocessor [11] has two floating point units, which are able to execute multiplication and addition in a single instruction (Fused Multiply Add or FMA) with a total peak performance of 18.8 Gflop/s. Furthermore, in order to achieve better memory bandwidth, IBM developed a prefetch engine. The prefetch engine consists of two parts, a stream filter and a stream prefetcher. The stream filter monitors the cache misses trying to guess data access patterns. The stream prefetcher prefetches data from memory based upon information given by the prefetch filter.

In MPIOM the computationally intensive subroutines make use of many 3D arrays and nested loops. The iteration space of the loops corresponds to the dimensions of the subdomains. Due to the fact that the arrays within a nested loop are too large to fit completely into cache, a lot of accesses to main memory occur. Memory bandwidth is the limiting factor for getting the most out the Power6 processor.

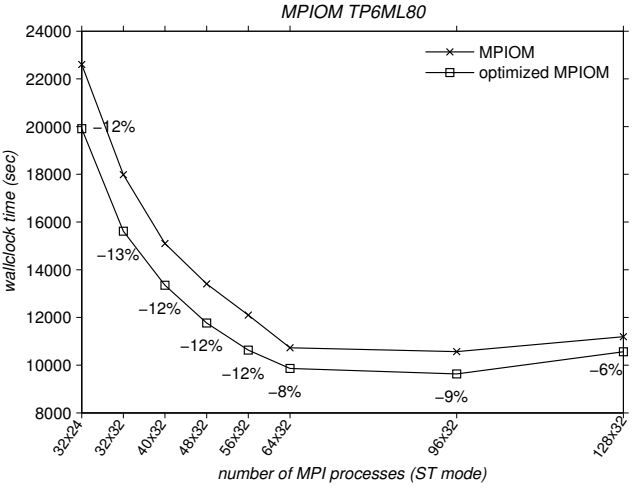



**Fig. 5.** Wallclock times of initial and optimized MPIOM model versions

Applying optimization techniques like loop tiling [5] or blocking algorithms [12] with the anticipation of improving cache locality, has proven to be the wrong way, since we saw no performance gain. Especially, when applying loop tiling in the zonal direction, a degradation in performance was noticed. Apparently, the prefetching mechanism is influenced negatively by these loop-transformations.

The Power6 core supports 16 parallel prefetch streams. We applied loop fusion in order to keep the number of streams at 16 and to achieve better reuse and predictability of the data access patterns. For the same reason, loops with more than 16 3D arrays were split. To eliminate conditionals (i.e. "if"-branches") inside of loops, arrays with indices satisfying the conditions were used. These indexing arrays were precomputed during model initialization phase. This allows loop unrolling by the compiler. Furthermore, expensive division operations were replaced by multiplication with the precomputed inverse.

All these modifications increased the FMA utilization from 37% to 39% . The maximum floating point operation ratio achieved is 3.47% of the peak performance on 768 processors, which is still very low. Nevertheless, the total wallclock time of MPIOM has been reduced by approx. 10%. Overall results are depicted in Fig. 5. Keeping in mind long simulation times, even a performance gain of a few percent proves to be worthwhile. As one would expect, these changes do not improve the scalability of the MPIOM model.

### 4.2 Reducing Communication

As mentioned in the previous section, communication overhead is a major bottleneck regarding efficiency of the code. Our analysis shows that improving the floating point operations ratio is not satisfying. Especially for a large number of processors, the dominating factor is the time spent in MPI communication,

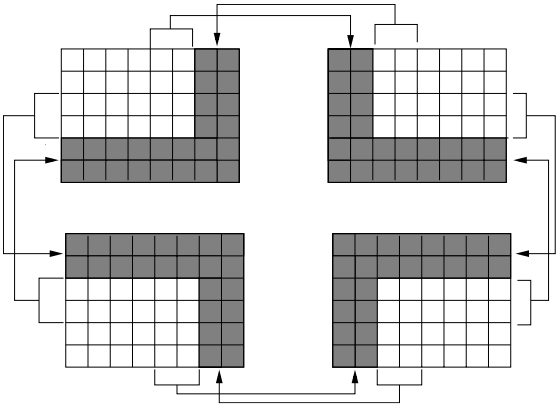


**Fig. 6.** Subdomains with two halo layers

which covers over 70% of total wall clock time. Consequently, the computation to communication ratio needs to be improved as well. A technique, which is used in grid based algorithms is expansion of the halo layer and has been applied to an ocean model according to [10]. The basic idea behind this technique is that the number of halo layers is increased, so that the number of columns communicated is increased.

Fig. 6 shows the case for two halo layers. The result is that on the one hand, the amount of communicated data is increased, but on the other hand the actual number of communication steps is decreased as detailed below.

As a first step, we applied this technique to the barotropic subsystem in MPIOM, which is numerically solved by red-black Successive-Over-Relaxation (SOR) method.

The SOR is an iterative splitting method for solving linear systems of equations. According to the red-black SOR algorithm, the grid points are separated into red and black points. In each iteration, first the new values of the red points are calculated. These are then communicated to the other processes. Afterwards, the calculations on the black points are performed, and the new values on the black points are communicated (see Fig. 7, left). The drawback is that communication takes place 2 times in each iteration.

Expanding the halo layer results in the following algorithm. Firstly, the red points are calculated, then the black points. The computations on the local subdomains are carried out until the outermost halo layer, thus including more columns (the extra halo layers in the interior of the subdomains). The time when the communication step occurs depends on the number of halo layers used, and must be half the number of halo layers. The algorithm is depicted in Fig. 7 (right).

The table 2 shows the results of the new SOR algorithm for a smaller MPIOM TP04L80 setup (cf. Table 1). The model integration was done on 64 processors. The simulated time period was 10 days. It is shown that increasing the number of halo layers decreases the time taken by the SOR solver. It can be seen that when

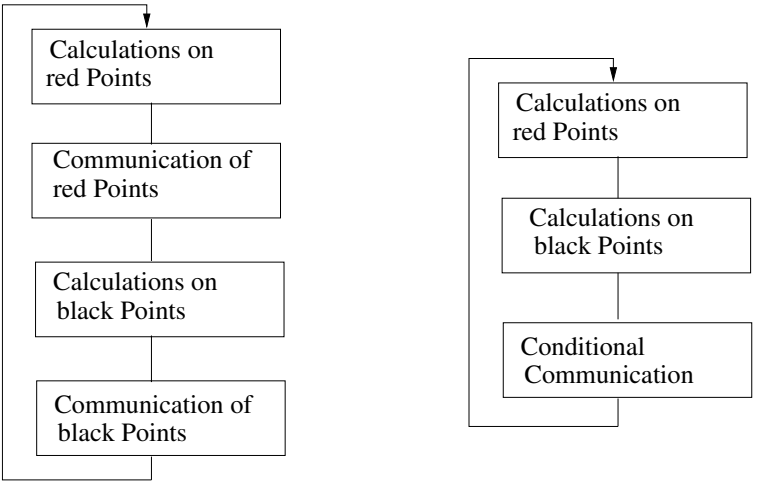



**Fig. 7.** Flowchart of original red-black SOR (left) and red-black SOR with expanded halos (right)

**Table 2.** Results of SOR solver for MPIOM TP04L80 model

| halo layers | communication step | wallclock time (seconds) |
|---|---|---|
| 1 | 1 | 27.6 |
| 2 | 1 | 15.6 |
| 4 | 2 | 13.9 |
| 8 | 4 | 8.8 |
| 10 | 5 | 9.2 |

using 2 halo layers, the performance gain is 43.47%. The maximum improvement of 68.1% has been achieved with 8 halo layers.

These results are very encouraging and the method of expanding halos is currently being applied to other parts of the code.

## 5 Summary and Outlook

In this paper we present the results of an investigation regarding the efficiency of climate models on current high performance computing platforms and the consequences for upcoming systems. Our performance analysis has shown that fundamental algorithmic changes are necessary in order to get the most out of the computational power available on supercomputers. Application of conservative optimizations (i.e. without algorithmic or structural changes) allow for performance gains in the order of 10%. Aggressive optimizations are often restricted by reevaluation of model physics.

The next optimization level includes statical and/or dynamical load balancing through use of alternative partitioning algorithms (e.g. general block decomposition, graph partitioning, space-filling curve partitioning etc.) or elimination of land points from partitions in the ocean model, optimization of communication, and implementation of parallel IO, that would lead to a significant performance and scalability improvements. For this reason, several projects (e.g. ScaleES[3], IS-ENES[4]) and new model development initiatives (ICON[5], HOMME[6], etc.) have been launched in the last years.

**Acknowledgements.** The authors would like to thank Mathias Puetz from IBM and Helmuth Haak from MPI-M for their valuable input regarding the code optimization. Thanks to Moritz Hanke, Thomas Jahns from DKRZ, and three anonymous reviewers for the recommendations and suggestions aimed at the improvement of the manuscript.

## References

1. Roeckner, E., Buml, G., Bonaventura, L., Brokopf, R., Esch, M., Giorgetta, M., Hagemann, S., Kirchner, I., Kornblueh, L., Manzini, E., Rhodin, A., Schlese, U., Schulzweida, U., Tompkins, A.: The atmospheric general circulation model ECHAM 5. PART I: Model description. MPI Report 349, Max Planck Institute for Meteorology, Hamburg, Germany, p. 127 (2003)
2. Marsland, S.J., Haak, H., Jungclaus, J.H., Latif, M., Roeske, F.: The Max-Planck-Institute global ocean/sea-ice model with orthogonal curvilinear coordinates. Ocean Modelling 5, 91–127 (2003)
3. Jungclaus, J.H., Botzet, M., Haak, H., Keenlyside, N., Luo, J.-J., Latif, M., Marotzke, J., Mikolajewicz, U., Roeckner, E.: Ocean circulation and tropical variability in the coupled model ECHAM5/MPI-OM. Journal of Climate 19, 3952–3972 (2006)
4. IPCC, 2007: Climate Change 2007: The Physical Science Basis. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change [Solomon, S., Qin, D., Manning, M., Chen, Z., Marquis, M., Averyt, K.B., Tignor, M., Miller H.L. (eds.)]. p. 996 .Cambridge University Press, Cambridge (2007)
5. Lam, M.S., Wolf, M.E.: A data locality optimizing algorithm. SIGPLAN Not. 39(4), 442–459 (2004), http://doi.acm.org/10.1145/989393.989437
6. The GRIB1/2 file format, http://www.wmo.ch/pages/index_en.html
7. The netCDF3/4 file format, http://www.unidata.ucar.edu/software/netcdf
8. The HDF5 file format, http://hdf.ncsa.uiuc.edu/HDF5
9. VampirTrace, http://www.tu-dresden.de/zih/vampirtrace
10. Beare, M.I., Stevens, D.P.: Optimisation of a parallel ocean general circulation model. Annales Geophysicae 15, 1369–1377 (1997)

[3] http://www.dkrz.de/dkrz/science/ScalES/ScalES
[4] https://is.enes.org
[5] http://www.icon.enes.org
[6] https://wiki.ucar.edu/display/homme/Home

11. Le, H.Q., Starke, W.J., Fields, J.S., O'Connell, F.P., Nguyen, D.Q., Ronchetti, B.J., Sauer, W.M., Schwarz, E.M., Vaden, M.T.: IBM POWER6 microarchitecture. IBM Journal Res. Dev. 51(6), 639–662 (2007)
12. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-D Blocking Optimization for Stencil Computations on Medern CPUs and GPUs. In: Proceedings of Supercomputing Conference SC 2010 (2010)

# Performances of Navier-Stokes Solver on a Hybrid CPU/GPU Computing System

Giancarlo Alfonsi, Stefania A. Ciliberti, Marco Mancini, and Leonardo Primavera

Fluid Dynamics Laboratory
Università della Calabria, Via Bucci 42b, 87036 Rende (Cosenza), Italy
{giancarlo.alfonsi,stefania.ciliberti,marco.mancini,
leonardo.primavera}@unical.it

**Abstract.** A computational code for the numerical integration of the incompressible Navier-Stokes equations for the execution of accurate calculations with the approach of the Direct Numerical Simulation (DNS), is implemented on a specially-assembled hybrid CPU/GPU computing system. The computational code is based on a mixed spectral-finite difference numerical technique, and is implemented onto the plane-channel computing domain, for the study of wall-bounded turbulence. The computing system includes one Intel Core *i7* (quad-core) processor, and two Nvidia C-1060 Tesla devices. High-resolution numerical simulations of the turbulent flow in the plane-channel domain are executed at wall-shear-velocity Reynolds number 200, and the performances of the code are reported in terms of parallel-machine metrics. Sample results of the simulations are also reported, in which some details are emphasized of the scientific information that have been obtained, mainly due to the high resolution at which the calculations have been executed, in virtue of the availability of such a powerful computing system.

**Keywords:** Navier-Stokes equations; Direct Numerical Simulation of turbulence; CPU/GPU hybrid computing systems.

## 1 Introduction

In fluid dynamics and turbulence research a wide class of methods of investigation involves numerical simulations. Numerical simulation of turbulence implies the execution of the numerical integration of the three-dimensional unsteady Navier-Stokes equations on an appropriate computing domain, for an adequate number of time steps. Different numerical techniques, ranging from finite differences, finite elements, spectral methods and appropriate combinations of the basic techniques into mixed techniques, can be used, besides different approaches to the modeling of turbulence (where modeling is needed, see among others, [1]).

One of the issues involved in these activities (at sufficiently high values of the Reynolds number) is the remarkable difference that exist between a *generic* solution of the Navier-Stokes equations and a solution of the same equations with the aim of obtaining a precise correlation of the results with turbulence physics. In the latter case the accuracy of the calculations has to be deeply monitored, and the approach to be followed is that of the Direct Numerical Simulation of turbulence (DNS).

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 404–416, 2011.

In the DNS approach the attitude of directly calculating all turbulent scales is followed, where the system of the governing equations is considered without modifications of any kind. The critical aspect in DNS is the accuracy of the calculations, that in theory should be as high as to resolve the Kolmogorov microscales in space and time (or at most limited multiples of them). As a consequence, in DNS the major difficulty in performing calculations at Reynolds numbers approaching those of practical interest, lies in the remarkable amount of computational resources required, and for a long time the outcome of this situation has been that only simple flow cases have been investigated numerically with DNS.

The advent of the supercomputing technologies has completely changed this scenario, opening new perspectives in the field of the high-performance computational fluid dynamics and turbulence simulation.

## 2 CPU/GPU Hybrid Architectures

In the last two/three decades, the computer technology has been dominated by microprocessors based on a single CPU (such as those of the Intel Pentium family or the AMD Opteron family), where the remarkable improvements continuously reached in the field of integrated-circuit technology were reflected in a yearly microprocessor-performance growth of about 35%. Mainly due to the cost advantages incorporated in mass-produced microprocessors, this performance growth-rate has allowed software systems of applicative nature to provide great functionalities to users, largely increasing the field of computer business (desktop and servers). However since about 2003, limits in the available instruction-level parallelism, memory latencies and also energy-consumption requirements, have slowed down and almost stopped the aforementioned growth process.

Thus, there has been a change in the electronic industry from high-performance single-microprocessor design, toward higher-performance multiple processor-per-chip design (the latter referred to as *processor cores*), so focusing the attention to the exploitation of thread-level parallelism and data-level parallelism. An immediate consequence of this change on the software-development community has been the need of adopting a different-than-before programming approach, consisting in particular in switching from implicitly-parallel programming, toward explicitly-parallel programming.

In the field of microprocessor architecture, one assists in the development of two main types, namely the *multicore* and the *manycore* processors (Figures 1 and 2).
A multicore processor (Figure 1) is designed to exploit massive quantities of on-chip resources in an efficient and scalable manner, combining each processor core (ALU, Arithmetic Logic Unit) with a switch, to create a modular element called a *tile*. An example is the Intel Core *i7* microprocessor, that includes four out-of-order processor cores, each supporting hyperthreading technology, and designed to maximize the speed of execution of sequential programs. Such a result has been made possible by the presence of a sophisticated control logic for sequential instructions, in which a large cache memory allows the reduction of instruction latencies and data- access latencies of large and complex applications.

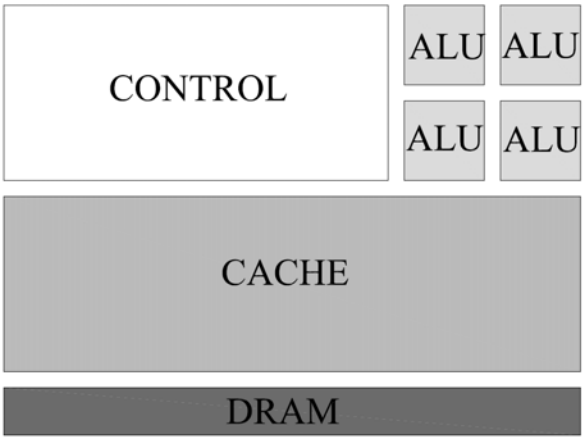



**Fig. 1.** Typical scheme of multicore CPU

A manycore processor (Figure 2) is characterized by many processors, each supporting many hardware threads, focusing more on the execution throughput of all threads, with a (small) cache memory bandwidth control on multiple threads that access the same memory data, not all needing to go to the DRAM. An example is the NVIDIA GTX-280 Graphic Processing Unit (GPU) with 240 cores, each of which is a massively multithreaded, in-order, single-instruction-issue processor, that shares its control and instruction cache with seven other cores ([2]).

While CPUs present intrinsic limitations in scaling serial performances (because of processor frequencies and power consumption), a significant interest has been put in massively-parallel programming of GPUs, being the latter specialized for intensive computing. The GPU is a highly explicit parallel environment, because it takes advantage of a large number of cores to improve calculation speed and management of huge datasets.

The key element that allows to synthesize the points of strength of both CPUs and GPUs is the NVIDIA's Computed Unified Device Architecture (CUDA) paradigm ([3]). CUDA has been introduced in 2007 to simplify the development of software applications and to promote the heterogeneous parallel-computing capabilities of hybrid CPU/GPU computing systems, towards massively-parallel programming based on coprocessing. The CUDA scalable programming model represents an extension of the C and C++ languages, and allows developers to use these languages as high-level programming tools. CUDA is based on three different levels of abstraction, namely, thread-groups hierarchy, shared memories, and barrier synchronization. These three elements provide the scalable parallel-programming structure.

The CUDA paradigm is based on three main features (see also at Figure 3):

- a (software) kernel, i.e. a serial program (or function) written for one (hardware) thread and designed to be executed by many threads;
- a thread block, defined as a set of threads that execute the same program and cooperate to compute a result;
- a grid, i.e. a set of thread blocks that independently execute in parallel the same kernel.

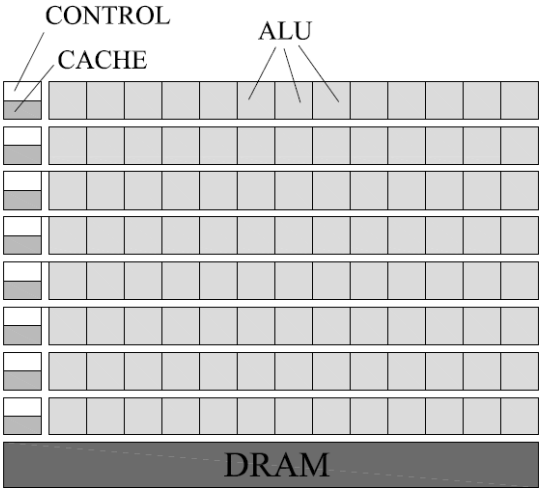



**Fig. 2.** Typical scheme of GPU

All the threads of a block may be synchronized directly by using a synchronization barrier. The latter guarantees that no thread in the block can proceed until all threads in the same block have reached the barrier. This is a remarkably-important task for communication between all threads in a block, and facilitate scalability. Another characteristic is that thread blocks may be executed independently. This guarantees that the thread blocks are scheduled in any order across any number of cores, in parallel or in series. Each thread has a private local memory for private variables. Each thread block has a shared memory, that is visible to all the threads of the same block, and that is used to initialize data in shared variables, compute results, and copy them into the global memory. All the threads can access the same global memory, that is used to read the input data and write the final results.

A typical CUDA program starts with a CPU (host) execution. When a kernel is launched, the execution is moved to a GPU (device) and a large set of threads are generated to exploit the data parallelism. This is the reason why the most heavy portion of the algorithm is addressed to the GPU. The NVIDIA C compiler (NVCC) separates the host activities from the device activities during the compilation process. The host code is written exclusively in ANSI C code, while the device code - organized in kernels - is written by using the extended ANSI C language, namely the CUDA C. The CUDA programming model also provides API functions for data transfer from the host memory to the allocated-device memory, and viceversa (each device has its own DRAM). The memory access is crucial for efficient CUDA programming (in terms of improvement of the memory bandwidth and overhead reduction). Thus, the global load/store instructions must coalesce individual parallel-threads requests from the same set of parallel threads (*warps*), into a single-memory block request, when the memory addresses fall in the same block and meet the alignment criteria.

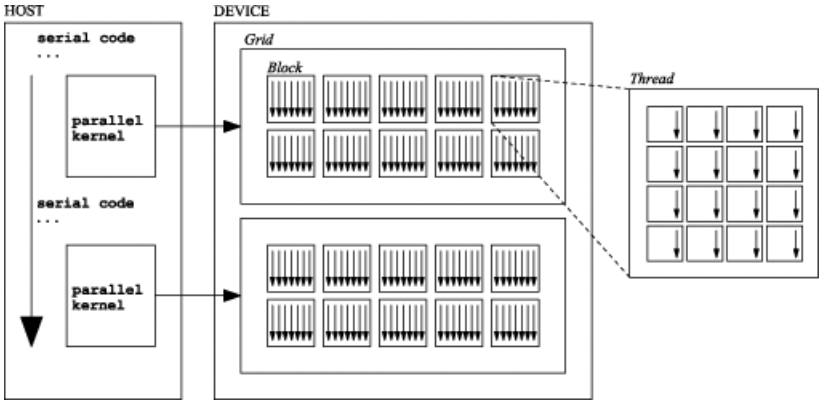



**Fig. 3.** Scheme of CUDA programming model

## 3 Numerical Technique

The system of the Navier-Stokes equations for incompressible fluids with constant properties is considered (in nondimensional conservative form, Einstein summation convention applies to repeated indices, *i=1,2,3*):

$$\frac{\partial u_i}{\partial t} + \frac{\partial}{\partial x_j}\left(u_i u_j\right) = -\frac{\partial p}{\partial x_i} + \frac{1}{Re_\tau}\frac{\partial^2 u_i}{\partial x_j \partial x_j} \tag{1}$$

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{2}$$

where $u_i$ are the velocity components, $p$ is the pressure, and $Re_\tau$ is the friction-velocity Reynolds number. The computational code for the solution of (1) and (2) that has been used for the present work is based on a mixed spectral-finite difference numerical technique, and is implemented onto the plane-channel computing domain for the study of wall-bounded turbulence (one can refer to [4] and [5] for further details on the numerical algorithm, and to [6] and [7] for past parallel implementations). The domain is considered homogeneous and periodic along $x$ and $z$ (the streamwise and spanwise directions, respectively), and the system of the governing equations is Fourier-transformed along those directions. The nonlinear terms of the momentum equation are calculated pseudo-spectrally, by anti-transforming the velocities back to physical space and performing the products. The 3/2 rule has been applied to avoid aliasing errors in transforming the results back to Fourier space. In order to capture the high velocity-gradients occurring near the walls, a grid-stretching law of hyperbolic-tangent type has been introduced within the spatial discretization along $y$ (the vertical direction). The spatial derivatives along $y$ have been evaluated by using a second-order centered finite difference scheme. For time advancement, a third-order explicit Runge-Kutta method has been implemented. For each Fourier mode (superscript ^), one has:

$$\frac{\hat{u}_i^{(l)} - \hat{u}_i^{(l-1)}}{\Delta t} = \alpha_l D\left(\hat{u}_i^{(l-2)}\right) + \beta_l D\left(\hat{u}_i^{(l-1)}\right) - \gamma_l A\left(\hat{u}_i^{(l-1)}\right) - \xi_l A\left(\hat{u}_i^{(l-2)}\right) - \left(\alpha_l + \beta_l\right)\frac{\partial \hat{p}}{\partial x_i} \tag{3}$$

where $l=1,2,3$ denotes each Runge-Kutta sub-step, $D$ and $A$ are the diffusive and advective terms, respectively, and $\alpha_l, \beta_l, \gamma_l, \xi_l$ assume constant values. The time advancement procedure is coupled with the fractional-step method. In (1), the pressure is interpreted as a projection operator, so that the velocity field and the pressure field are decoupled. At each sub-step $l$ and for each Fourier mode $i$, an intermediate velocity field is introduced (superscript *):

$$\frac{\hat{u}_i^{*(l)} - \hat{u}_i^{(l-1)}}{\Delta t} = \alpha_l D\left(\hat{u}_i^{(l-2)}\right) + \beta_l D\left(\hat{u}_i^{(l-1)}\right) - \gamma_l A\left(\hat{u}_i^{(l-1)}\right) - \xi_l A\left(\hat{u}_i^{(l-2)}\right) \tag{4}$$

and the pressure is used to project the intermediate-velocity field into the divergence-free space by solving the Poisson problem:

$$\nabla^2 \hat{p}^{(l)} = \frac{1}{\Delta t\left(\alpha_l + \beta_l\right)}\left(\frac{\partial \hat{u}^{*(l)}}{\partial x} + \frac{\partial \hat{v}^{*(l)}}{\partial y} + \frac{\partial \hat{w}^{*(l)}}{\partial z}\right) \tag{5}$$

With the pressure obtained from the solution of (5), it is possible to evaluate the final values of the velocity field, to obtain the next update of velocity and pressure:

$$\frac{\hat{u}_i^{(l)} - \hat{u}_i^{*(l)}}{\Delta t} = -\left(\alpha_l + \beta_l\right)\frac{\partial \hat{p}^{(l)}}{\partial x_i} \tag{6}$$

No-slip boundary conditions at the walls and periodic conditions in the streamwise and spanwise directions have been applied to the velocity, while a Neumann-type boundary condition has been used for the pressure.

## 4 GPU Implementation of Navier-Stokes Solver

The analysis of the numerical method described in the previous Section leads to a novel parallel implementation of the Navier-Stokes solver on the GPU, based on the following main steps ( $N_x, N_y, N_z$ are the grid points along, respectively, *x,y* and *z*):

- computation of the intermediate velocity field $\hat{u}_i^{*(l)}$, once the velocity field $\hat{u}_i^{*(l-1)}$ in spectral space is given. For each Runge-Kutta sub-step, 2D FFTs are applied along the *x*- and *z*-axes, for the evaluation the nonlinear terms of equation (2);
- computation of the pressure field $\hat{p}^{(l)}$, once the intermediate velocity field $\hat{u}_i^{*(l)}$ is given. This phase requires the implementation of the finite-difference algorithm along the *y*-direction, for the solution of the Poisson pressure problem;
- updating of the velocity field $\hat{u}_i^{(l)}$ by implementing the third-order Runge-Kutta algorithm.

The program, written in CUDA 3.2, starts with the CPU execution. The host reads an input file, containing data about an initial velocity field. The host is also devoted to the memory management of the complex velocity field $\hat{u}_i$ in Fourier space, and all the related data structures, by calling a set of kernels. They provide functions to allocate, deallocate and copy device memory. In particular, the data structures are allocated using **cudaMalloc( )** and stored in the memory as shown in Figure 4, while data transfer of *u,v,w* from the host (**h_u, h_v, h_w**) to the device (**d_u, d_h, d_w**) memory, are managed by **cudaMemcpy( )** (see at Listing 1).

```
cudaMalloc();
cudaMemcpy(d_u, h_u,cudaMemcpyHostToDevice, size);
cudaMemcpy(d_v, h_v,cudaMemcpyHostToDevice, size);
cudaMemcpy(d_w, h_w,cudaMemcpyHostToDevice, size);
```

**List. 1.** Scheme of data-structures allocation and host-to-device transfer

The GPU is devoted to execute the core of the algorithm. For a given velocity field stored on the memory device, batched 2D complex-to-real (C2R) and real-to-complex (R2C) Fourier transforms have been implemented, using the CUFFT library provided by NVIDIA [8]. In particular, an in-place batched 2D/R2C/FFT has been used in order to transform the velocity field from spectral to physical space, using a batch of $N_y$ planes ( $x-z$ planes). Then, each product between the mutual components of the velocity field is computed by a kernel, named product_kernel. For each product $u\hat{u}, u\hat{v}, u\hat{w}, v\hat{v}, v\hat{w}, w\hat{w},$ an in-place batched 2D/C2R/FFT is executed, in order to transforming those quantities back to spectral space (see at Listing 2).

For each Runge-Kutta sub-step, it is now possible to calculate both the diffusive (*D*) and the advective (*A*) terms of equation (3), by using two appropriately-designed

```
cufftExecC2R(batchBackPlan,(cufftComplex *)d_u,
(float *)d_u);
cufftExecC2R(batchBackPlan,(cufftComplex *)d_v,
(float *) d_v);
cufftExecC2R(batchBackPlan,(cufftComplex *)d_w,
(float *) d_w);
product_kernel <<< blocksPerGrid,threadsPerBlock >>>
(d_u, d_v, d_w,d_uu, d_uv, d_uw,d_vv, d_vw, d_ww);
cufftExecR2C(batchFwdPlan,(float *)d_uu,(cufftComplex *)
d_uu);
......
cufftExecR2C(batchFwdPlan,(float *)d_ww,(cufftComplex *)
d_ww);
```

**List. 2.** Scheme of FFT operations for the solution of non-linear terms

kernels, the **diffusive_kernel** and the **advective_kernel**, to minimize memory access using memory coalescing and increasing the computation-to-memory ratio. The intermediate velocity field is calculated using another kernel, named **velstar_kernel**. Once all the components of the velocity field and all the components of the diffusive and advective terms are given, the three components of the intermediate velocity field (**d_ustar**, **d_vstar**, **d_wstar**) of equation (4) are computed (see at Listing 3).

```
advective_kernel <<< blocksPerGrid,threadsPerBlock >>>
(d_uu, d_uv, d_uw,d_vv, d_wv, d_ww,d_nu, d_nv, d_nw);
diffusive_kernel <<< blocksPerGrid,threadsPerBlock >>>
(d_u,d_v,d_w,d_lu,d_lv,d_lw);
velstar_kernel <<< blocksPerGrid,threadsPerBlock >>>
(d_u,d_v,d_w,d_nu,...,d_lu,...,d_ustar, d_vstar,d_wstar);
```

**List. 3.** Scheme for the calculation of the intermediate velocity field

As mentioned before, $\hat{u}_i^{*(l)}$ does not satisfy the incompressibility constraint, so that it becomes necessary to implement the fractional-step method. This is done by implementing a set of kernels that compute the rhs of equation (5), once the intermediate velocity field is given, and solve the Poisson problem. In order to solve the Poisson problem, a total of ( $N_x \times N_z$ ) linear (tridiagonal) systems are solved on the GPU, using the Thomas algorithm. The sequential counterpart of the Thomas algorithm is redesigned, to keep the values in the GPU registers, implementing a memory-coalesced data-access mechanism. Thus, each thread solves a tridiagonal system by using a modified Thomas algorithm for CUDA. As concerns the principal diagonal (that is stored on the device), memory coalescing of load/store instructions is guaranteed for the threads that belong to the same warp, in virtue of the thread-index mapping. A good degree of memory optimization is also achieved by storing the lower- and upper diagonals in the constant memory (see at Listing 4).

```
tn_kernel <<< blocksPerGrid,threadsPerBlock >>>
(rk,h_dt,d_tn,d_ustar, d_vstar, d_wstar);
solve_poisson_kernel <<< blocksPerGrid, threadsPerBlock >>>
(d_dpri,d_tn,d_p,d_work);
```

**List. 4.** Scheme for the solution of the Poisson problem

After the pressure component is computed, a kernel for updating of the velocity field (**update_velocity_kernel**) is executed, according to equation (6). Then, a device-to-host data transfer of the computed values is performed, for both the execution of a new iteration and to save the data into the database (Listing 5).

```
update_velocity_kernel <<< blocksPerGrid,threadsPerBlock >>>
(d_ustar,d_vstar,d_wstar,d_p,rk,h_dt,d_u, d_v, d_w);
cudaMemcpy(h_u,d_u,cudaMemcpyDeviceToHost,size);
cudaMemcpy(h_v,d_v,cudaMemcpyDeviceToHost,size);
cudaMemcpy(h_w,d_w,cudaMemcpyDeviceToHost,size);
```

**List. 5.** Scheme for the updating of the velocity field and for the device-to-host data transfer

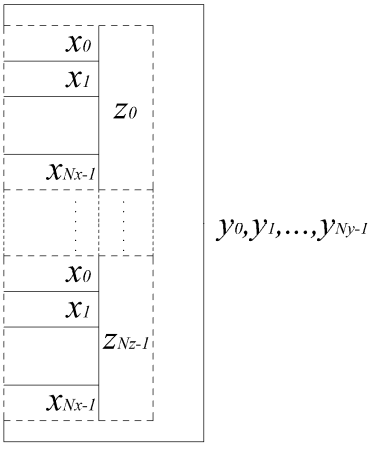



**Fig. 4.** Data-structure memorization scheme

## 5 Computing System

The numerical simulations have been executed on a specially-assembled hybrid CPU/GPU computing system (see at Figure 5 for a picture of the system), that includes a motherboard ASUS P6T7-WS SuperComputer, an Intel Core *i7* processor at 2.66 GHz of processor-core clock, 12 GB of DDR3 RAM, and two NVIDIA Tesla C1060 boards, based on NVIDIA CUDA technology. Each Tesla board can handle 933 GFLOP/s of single-precision floating-point processing, is equipped with 4 GB of GDDR3 memory at 102 GB/s bandwidth, and contains 30 multiprocessors, each with 8 scalar single-precision floating-point processor cores, 1 double-precision floating-point unit and 16 kB shared memory for threads cooperation (Figure 6). The total number of cores is 240, at 1.3 GHz of processor-core clock. The system is also equipped with a NVIDIA GeForce GTX 285 with 1 GB of GDDR3 memory at 159 GB/s bandwidth, while the total number of cores is 240, at 648 MHz of processor-core clock. The GeForce is mainly used for visualization.

As concerns storage, the system is equipped with 5 Western Digital VelociRaptor 300 GB SATA hard drives (at 10000 rpm) and 1 Seagate Barracuda 1 TB SATA hard drive (at 7200 rpm).

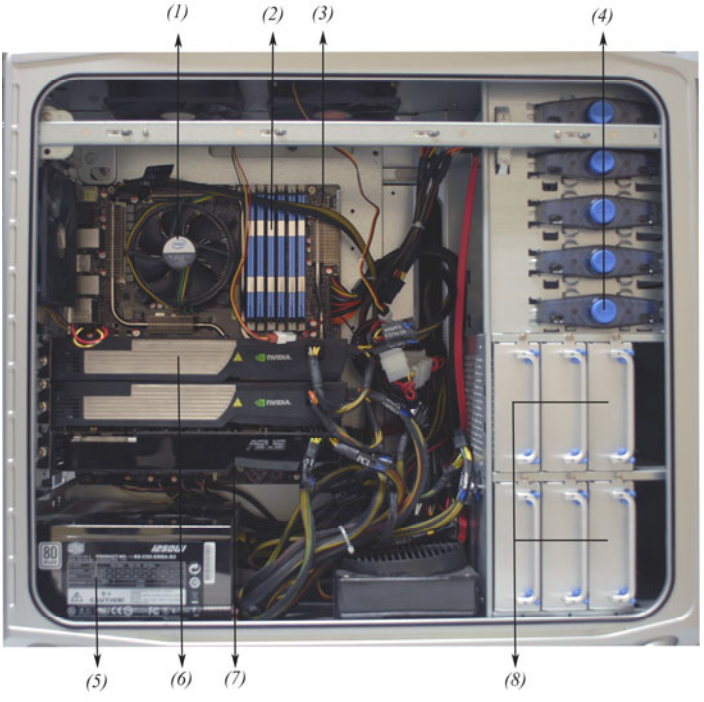



**Fig. 5.** A picture of the computing system used for the calculations: *(1)* 1 Intel *i7* (quad-core) Core (behind fan); *(2)* 12 GB DDR3 RAM; *(3)* 1 Asus P6T7 WS SuperComputer motherboard; *(4)* 5 DVD drives; *(5)* 1 1250 W power supply; *(6)* 2 NVIDIA C-1060 Tesla devices; *(7)* 1 NVIDIA GeForce GTX 285 video card; *(8)* 5 Western Digital 300 GB (10000 rpm) VelociRaptor hard drives + 1 Seagate 1 TB (7200 rpm) Barracuda hard drive

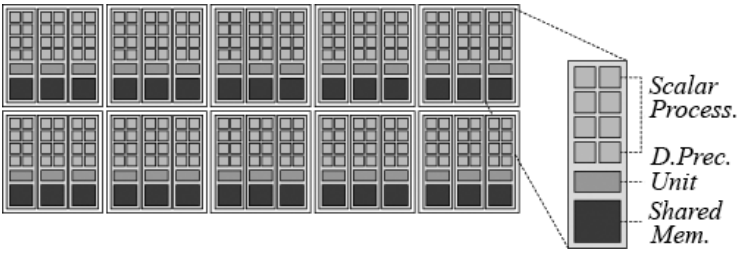



**Fig. 6.** Scheme of Tesla-board architecture

## 6 Code Performances

The numerical simulations have been executed on a computational grid that includes 256 grid points along the streamwise direction ($x$), 181 grid points along the vertical direction ($y$), and 256 points along the spanwise direction ($z$), while the nondimensional time step was $\Delta t^+ = 10^{-4}$. The nondimensional values of the grid

spacing were $\Delta x^+ = 9.82$, $\Delta y^+ = 0.25$ (at the wall), $\Delta y^+ = 3.87$ (at channel center), and $\Delta z^+ = 4.91$. These values have to be compared with those of the nondimensional Kolmogorov length microscale $\eta^+ = 1.89$ and with the nondimensional Kolmogorov time microscale $\tau_\eta^+ = 3.59$ ($x_i^+ = x_i u_\tau / \nu$; $t^+ = t u_\tau^2 / \nu$). The initial transient of the flow in the channel has been first simulated, the turbulent statistically-steady state has been reached, and then simulated for 50000 nondimensional time steps $\Delta t^+$. One-hundred nondimensional flow-field instants have been recorded, one every 500 $\Delta t^+$.

In order to evaluate the performance of the Navier-Stokes solver, the CPU/GPU implementation has been compared with a sequential version (1 CPU) and a OpenMP version (2 and 4 CPUs). The parallel performance has been evaluated by measuring the overall code execution time, focusing on the advective-, diffusive- and Poisson problem- execution time at each Runge-Kutta step. The measured times do not include the I/O operations. The parallel performance of the computational code has been evaluated by using the speedup $S$, defined as:

$$S = \frac{T_1}{T_{pr}} \tag{7}$$

where $T_1$ is the execution time of the sequential algorithm and $T_{pr}$ is the execution time of the parallel algorithm on $pr$ processors. As concerns the code CPU/GPU implementation, the speedup is defined as the ratio between the total execution time on a CPU ($T_{CPU}$) and that on a GPU ($T_{GPU}$):

$$S = \frac{T_{CPU}}{T_{GPU}} \tag{8}$$

The results that have been obtained are reported in Figure 7, while the (absolute) execution times are shown in Table 1 (the *Total Time* at the 5th column of Table 1, also includes also the data transfer from the device to the host). From both Figure 7 and Table 1 one can easily notice how the CUDA solver significantly outperforms the different CPU implementations.

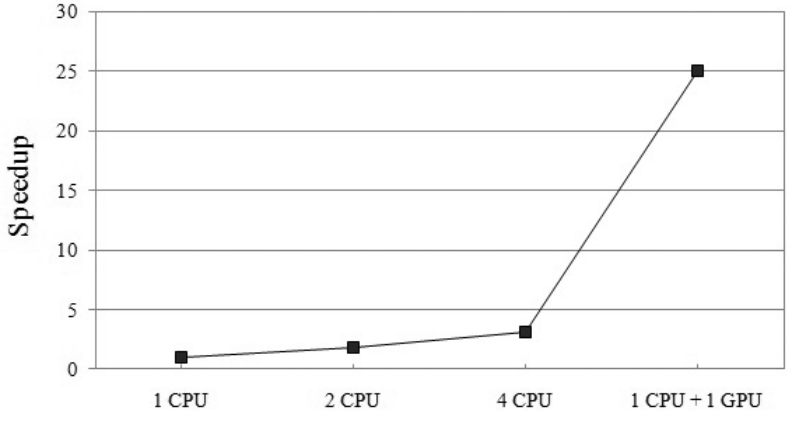



**Fig. 7.** Speedup of the calculations (full time step)

**Table 1.** Execution times for one time step of the calculation process (s)

| *Processing Unit* | *Convective Term (s)* | *Diffusive Term (s)* | *Poisson Problem (s)* | *Total Time (s)* |
|---|---|---|---|---|
| 1 CPU | 4.923 | 0.342 | 0.801 | 7.647 |
| 2 CPU | 2.592 | 0.174 | 0.402 | 4.070 |
| 4 CPU | 1.542 | 0.096 | 0.204 | 2.430 |
| 1 CPU + 1 GPU | 0.02628 | 0.02073 | 0.03600 | 0.30741 |

## 7 Concluding Remarks

In this work, the issue of the performances of a computational code for the numerical integration of the Navier-Stokes equations on a hybrid CPU/GPU computing system has been addressed. It has been possible to execute the accurate numerical simulations of the turbulent flow of a incompressible fluid in a plane channel at friction-velocity Reynolds number $Re_\tau = 200$, by using 1 GPU. Near-future work will involve a multi-node/multi-GPU implementation of the code, to reach higher values of $Re_\tau$.

The Direct Numerical Simulation of turbulence represents the only possible approach for the rigorous investigation of turbulence physics with numerical means. In fact, the most effective use of the different techniques available for the identification of the turbulent-flow structures in any kind of turbulent flow, typically relies on the analysis of huge three-dimensional, time-dependent turbulent-flow databases, nowadays necessarily of High-Performance DNS origin. In this context, the use of most advanced high-performance computing achitectures and procedures in the numerical simulation of turbulence appear as a not-avoidable option.

## References

1. Alfonsi, G.: Reynolds-Averaged Navier-Stokes equations for turbulence modeling. Appl. Mech. Rev. 62, 40802-1– 40802-20 (2009)
2. Kirk, D., Hwu, W.W.: Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, San Francisco (2010)
3. NVIDIA CUDA C Programming Guide. Version 3.2 (2010), `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`
4. Alfonsi, G., Passoni, G., Pancaldo, L., Zampaglione, D.: A spectral-finite difference solution of the Navier-Stokes equations in three dimensions. Int. J. Num. Meth. Fluids 28, 129 (1998)
5. Passoni, G., Alfonsi, G., Galbiati, M.: Analysis of hybrid algorithms for the Navier-Stokes equations with respect to hydrodynamic stability theory. Int. J. Num. Meth. Fluids 38, 1069 (2002)

6. Passoni, G., Alfonsi, G., Tula, G., Cardu, U.: A wavenumber parallel computational code for the numerical integration of the Navier-Stokes equations. Parall. Comput. 25, 593 (1999)
7. Passoni, G., Cremonesi, P., Alfonsi, G.: Analysis and implementation of a parallelization strategy on a Navier-Stokes solver for shear flow simulations. Parall. Comput. 27, 1665 (2001)
8. NVIDIA CUDA CUFFT Library. Version 3.2 (2010), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUFFT_Library.pdf

# Optimal Design of Multi-product Batch Plants Using a Parallel Branch-and-Bound Method

Andrey Borisenko[1], Philipp Kegel[2], and Sergei Gorlatch[2]

[1] Tambov State Technical University, Russia
borisenko@mail.gaps.tstu.ru
[2] University of Muenster, Germany
{philipp.kegel,gorlatch}@uni-muenster.de

**Abstract.** In this paper we develop and implement a parallel algorithm for a real-world application: finding optimal designs for multi-product batch plants. We describe two parallelization strategies – for systems with shared-memory and distributed-memory – based on the branch-and-bound paradigm and implement them using OpenMP (Open Multi-Processing) and MPI (Message Passing Interface), correspondingly. Experimental results demonstrate that our approach provides competitive speedup on modern clusters of multi-core processors.

**Keywords:** multi-product batch plant, parallel optimization, branch-and-bound, master-worker, global optimization, MPI, OpenMP.

## 1 Motivation and Related Work

Selecting the equipment of a Chemical-Engineering System (CES) is one of the main problems when designing chemical multi-product batch plants, e.g., for synthesizing chemical dyes and intermediate products, photographic materials, pharmaceuticals etc. A solution of this problem comprises finding the optimal number of devices at processing stages, as well as working volumes or areas of working surfaces of each of these devices. Working volumes and the areas of working surfaces are chosen from a discrete set of standard values. One needs to find an optimal combination of equipment variants using a criterion of optimality, for example, the minimal total capital equipment costs.

The problem of optimal design of multi-product batch plants is a mixed integer nonlinear programming (MINLP) problem [5, 15]. Existing techniques – Monte Carlo method, genetic algorithms, heuristic methods etc. – allow for obtaining suboptimal solutions. Performing an exhaustive search (pure brute-force solution) for finding a global optimum is usually impractical because of the large dimension of the problem. For example, in our earlier work [9], a CES consisting of 16 stages is presented where each process stage can be equipped with devices of 5 to 12 standard sizes. Thus, the number of choices in this case is $5^{16}$–$12^{16}$ (which is approximately $10^{11}$–$10^{17}$).

In this paper, we explore the possibility of accelerating the calculations for finding optimal CES designs using a parallelized branch-and-bound algorithm.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 417–430, 2011.

Branch-and-bound is a one of the most popular techniques used for solving optimization problems in various fields (e.g., combinatorial optimization, artificial intelligence, etc.). It is also used to solving MINLPs [8]. Branch-and-bound uses a queue of subproblems obtained by decomposing the original problem: it systematically enumerates all solutions and discards a large number of them by using upper and lower bounds of their objective function [3]. In branch-and-bound, the search space is usually considered as a tree, which allows for a structured exploration of the search space. Calculations for the various branches can be carried out simultaneously, which is used to create a parallel version of this method.

Parallel branch-and-bound algorithms have been discussed extensively in the literature. Parallel formulations of depth-first branch-and-bound search are presented in [7]. Martí et al. propose a branch-and-bound algorithm and develop several upper bounds on the objective function values of partial solutions for the Maximum Diversity Problem (MDP) [11]. Mansa et al. analyze the performance of parallel branch-and-bound algorithms with best-first search strategy by examining various anomalies on the expected speed-up [10]. In [6], Gendron et al. present several strategies to exploit parallelism using examples taken from the literature and show that the choice of strategy is greatly influenced by the parallel machine used, as well as by the characteristics of the problem. Rasmussen et al. solve discrete truss topology optimization problems using a parallel implementation of branch-and-bound [16]. In [18], Reinefeld et al. compare work-load balancing strategies of two depth-first searches and propose a scheme that uses fine-grained fixed-sized work packets. Sanders et al. [19] introduce randomized dynamic load balancing algorithms for tree-structured computations, a generalization of backtrack search. Aida [1] et al. discuss the impact of the hierarchical master-worker paradigm on the performance of solving an optimization problem by a parallel branch-and-bound algorithm on a distributed computing system. Bouziane et al. [2] propose a generic approach to embed the master-worker paradigm into software component models and describes how this generic approach can be implemented within an existing software component model. Cauley et al. [4] present a detailed placement strategy designed to exploit distributed computing environments, where the additional computing resources are employed in parallel to improve the optimization time. A Mixed Integer Programming (MIP) model and branch-and-cut optimization strategy are employed to solve the standard cell placement problem. In [21], Zhou at al. present a parallel algorithm for enumerating chemical compounds, which is a fundamental procedure in Chemo- and Bio-informatics.

The problem of optimal design of multi-product batch plants is also covered in the literature. Moreno at al. developed a novel linear generalized disjunctive programming (LGDP) model for the design of multi-product batch plants optimizing both process variables and the structure of the plant through the use of process performance models [13]. Rebennack at al. [17] present a mixed-integer nonlinear programming (MINLP) formulation, where non-convexities are due to the tank investment cost, storage cost, campaign setup cost and variable production rates. The objective of the optimization model is to minimize the sum of

the production cost per ton per product produced. In [20], Wang at al. present a framework for the design and optimization of multi-product batch processes under uncertainty with environmental considerations.

In this paper, we develop a parallel branch-and-bound algorithm for the globally optimal design of real-world multi-product batch plants, and implement it on modern clusters of multi-core processors.

## 2 Problem Formulation

A chemical-engineering system (CES) is a set of equipment (reactors, tanks, filters, dryers etc.) which implement the processing stages for manufacturing certain products. Assuming that each processing stage is equipped with a single device, the problem can be formulated as follows:

A CES consists of a sequence of $I$ processing stages. Each processing stage of the system can be equipped with a device from a finite set $X_i$, with $J_i$ being the number of device variants in $X_i$. All device variants of a CES are described as $X_i = \{x_{i,j}\}$, $i = \overline{1,I}, j = \overline{1,J_i}$, where $x_{i,j}$ is the main size $j$ (working volume, working surface, etc.) of the device suitable for processing stage $i$.

Each variant $\Omega_e$, $e = \overline{1,E}$ of a CES, where $E = \prod_{i=1}^{I}(J_i)$ is the number of all possible system variants, is an ordered set of devices work sizes, selected from the respective sets. For example, for a system with 3 processing stages ($I = 3$), the first stage may be equipped with devices selected from a set of 2 working sizes, i. e. $J_1 = 2, X_1 = \{x_{1,1}, x_{1,2}\}$, the second stage from 3 working sizes $J_2 = 3, X_2 = \{x_{2,1}, x_{2,2}, x_{2,3}\}$, and the third stage from 2 working sizes $J_3 = 2, X_3 = \{x_{3,1}, x_{3,2}\}$. Hence, the number of all possible system variants is given by $E = J_1 \cdot J_2 \cdot J_3 = 2 \cdot 3 \cdot 2 = 12$.

As the order of processing stages is predefined, some system variants, e. g., $\{x_{1,1}, x_{2,1}, x_{3,2}\}$, $\{x_{1,2}, x_{2,1}, x_{3,1}\}$ are valid, but others, e. g., $\{x_{3,1}, x_{2,1}, x_{1,2}\}$, $\{x_{2,2}, x_{3,1}, x_{1,1}\}$ are not. Each variant $\Omega_e$ of a system should be in operable condition (*compatibility constraint*), i. e. it should satisfy the conditions of a joint action for all its processing stages: $S(\Omega_e) = 0$.

An operable variant of a CES should run at a given production rate in a given period of time (*processing time constraint*), such that it satisfies the restrictions for the duration of its operating period $T(\Omega_e) \leq T_{max}$, where $T_{max}$ is a given maximum period of time.

Thus, designing a multi-product batch plant can be stated as the following optimization problem: to find a variant $\Omega^* \in \Omega_e$, $e = \overline{1,E}$ of a CES, where the optimality criterion – equipment costs $Cost(\Omega_e)$ – reaches a minimum and both compatibility constraint and processing time constraint are satisfied:

$$\Omega^* = argmin\ Cost(\Omega_e), \Omega^* \in (\Omega_e),\ e = \overline{1,E} \tag{1}$$

$$\Omega_e = \{(x_{1,j_1}, x_{2,j_2}, \ldots, x_{I,j_I}) | j_i = \overline{1,J_i}, i = \overline{1,I}\}, e = \overline{1,E} \tag{2}$$

$$x_{i,j} \in X_i,\ i = \overline{1,I}, j = \overline{1,J_i} \tag{3}$$

$$S(\Omega_e) = 0,\ e = \overline{1, E} \tag{4}$$

$$T(\Omega_e) \le T_{max},\ e = \overline{1, E} \tag{5}$$

In this paper, we use the comprehensive mathematical model of CES operation, including expressions for checking constraints, calculating the optimization criterion, etc., which was initially presented in [9].

## 3 Sequential Implementation and Its Optimization

In this section, we describe the sequential implementation of a branch-and-bound algorithm for finding an optimal CES.

All possible variants of a CES with $I$ stages can be represented by a tree of height $I$ (see Figure 1). Each level of the tree corresponds to one processing stage of the CES. Each edge corresponds to a selected device variant taken from set $X_i$, where $X_i$ is the set of possible device variants at stage $i$ of the CES. For example, the edges from level 0 of the tree correspond to elements of $X_1$. Each node $n_{i,k}$ at the tree layer $N_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,k}\}$, $i = \overline{1, I}, k = \overline{1, K_i}, K_i = \prod_{l=1}^{i}(J_l)$ corresponds to a variant of a beginning part of the CES, composed of devices for stages 1 to $i$ of the CES. Each path from the tree's root to one of its leaves thus represents a complete variant of the CES.

To enumerate all possible variants of a CES in the aforementioned tree, a depth-first traversal is performed: starting at level 0 of the tree, all device variants

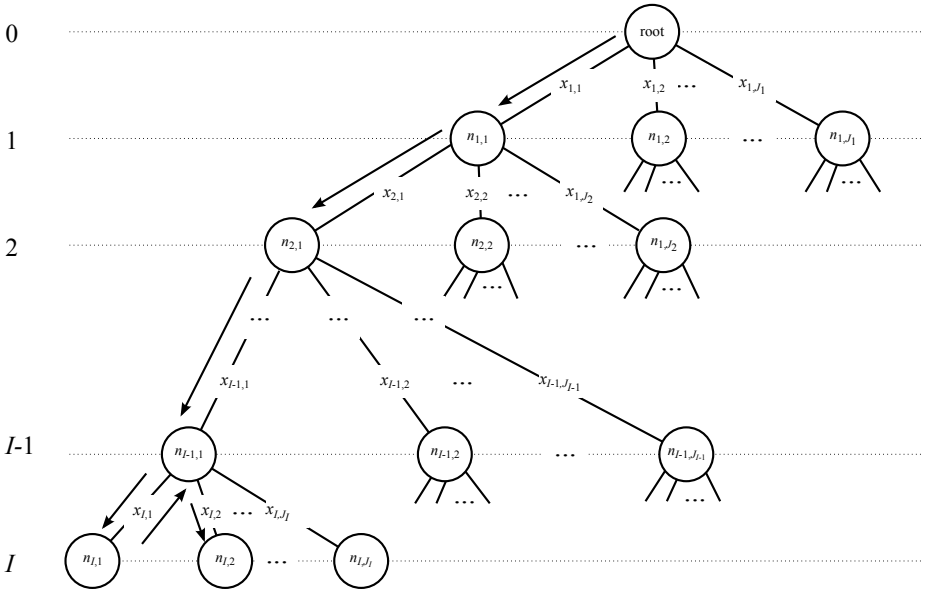



**Fig. 1.** Tree traversal in depth-first search

```
1  FindSolution() {    EnumerateVariants(0);   }
2
3  /* recursive tree traversal */
4  EnumerateVariants(level) {
5    if (level < I) {
6      for (j = 1; j <= J[level]; j++) {
7        /* append device variant to beginning part */
8        W[level] = X[level, j];
9        /* check compatibility constraint and upper bound */
10       if (S(W) == 0 && PartCost(W, level) < minCost) {
11         /* search recursively */
12         EnumerateVariants (level + 1);  }   }   }
13   else { /* leaf node */
14     /* check processing time constraint */
15     if (T(W) <= Tmax) {
16       /* check optimality criterion */
17       if (Cost(W) < minCost) {
18         /* make current solution new optimal solution */
19         Wopt = W;
20         minCost = Cost(Wopt);   }   }   }
21 }
```

**Listing 1.** Sequential implementation of branch-and-bound.

of the CES at a given level are enumerated and appended to the valid beginning parts of the CES. Valid beginning parts are obtained at previous levels, starting with an empty beginning part at level 0. This process continues recursively for all valid beginning parts that result from appending device variants of the current level to the valid beginning parts from previous levels. When a leaf node is reached, the recursive process stops and the current solution is compared to the current optimal solution, possibly replacing it.

Since a complete tree traversal (selecting a device on each edge traversal) and checking constraints (see Equations 4 and 5) would result in considerable computational costs, we use the branch-and-bound technique, with pseudo-code shown in Listing 1. If not stated otherwise, the names of variables correspond to the names in the problem formulation (see Section 2). The tree traversal starts by calling procedure `EnumerateVariants` at level 0 (line 1). This method continues recursively until the optimal CES `Wopt` has been found. Here, `Wopt` is a vector of length `I`, specifying the device variant at each stage of the optimal solution. When traversing the tree, the compatibility constraint (see Equation 4, function `S()`) is checked for the corresponding part of the CES. In addition, we compare the cost for the current beginning part of the CES, consisting of the first `level` stages (function `PartCost()`) with a global upper bound (variable `minCost`). The initialization of the upper bound is done as sum of all maximum device costs for each productions stage. If the current beginning part of the CES fulfills the compatibility constraint and its costs do not exceed the global upper bound (line 10), we recursively continue tree traversal to the next level (`EnumerateVariants(level + 1)`, line 12). Otherwise we discard deeper levels of the tree and backtrack to the previous level. If a leaf node of the tree is reached (line 13 ff.), the processing time constraint (see Equation 5, function `T()`) is checked for the corresponding CES (line 15). If this constraint is fulfilled, a new solution has been found and its costs (Equation 1, function `Cost()`) are compared to the cost of the last known optimal solution (line 17). If a better

**Table 1.** Averaged execution times of the various algorithm parts

| Algorithm part | Execution time ($\mu$s) |
|---|---|
| Recursive call of `EnumerateVariants()` | 0.1 |
| `S(W)` | 4.0 |
| `PartCost(W,level)` | 0.3 |
| `T(W)` | 417.0 |
| `Cost(W)` | 0.7 |

```
12  ...
13  else { /* leaf node */
14    /* check optimality criterion */
15    if (Cost(W) < minCost) {
16      /* check processing time constraint */
17      if (T(W) <= Tmax) {
18        /* make current solution new optimal solution */
19        Wopt = W;
20        minCost = Cost(Wopt);   }   }   }
21  ...
```

**Listing 2.** Optimizing the sequential algorithm by swapping checks of processing time constraint (slow) and optimality criterion (fast).

solution is obtained, it replaces the previous optimal solution and its costs are taken as new upper bound (line 19–20).

We developed a C++-based implementation of the presented sequential algorithm to perform runtime experiments. As a test case we used the calculation of a CES consisting of 16 processing stages (`I = 16`) with 5 device variants at every stage. Our experiments were conducted on a system comprising 2 Intel Westmere processors (X5650, 6 cores, running at 2.6 GHz) and 4 GB RAM. We use the Intel C++ Compiler version 11.1. We evaluated the execution times of the algorithm's parts to identify the most expensive of them. From the averaged experimental results (Table 1) for our sequential implementation, we observe that the most expensive operation is calling of `T(W)` for checking the processing time constraints of the CES.

The runtimes presented in the table are quite small for a single computation. But in the searching process with multiple repetitions (billions times) they can add up to tens and hundreds of hours. For our example (16 processing stages with 5 devices variants each), the overall runtime is 27h 11m. In order to reduce the algorithm's runtime, the number of calls of function `T(W)` has to be minimized.

We have implemented the following optimization of the sequential program. From Table 1 we deduce that checking the optimality criterion (`Cost(W)`) is a comparatively cheap operation. If we execute this operation as early as possible, we can discard suboptimal solutions without checking the processing time constraint which is a rather expensive operation. Therefore, we modify the algorithm by swapping the checks for the optimality criterion and the processing time constraint (see Listing 2, lines 15, 17)

To evaluate the performance impact of our optimization, we repeat our measurements for the modified implementation using the aforementioned experimental setup. Our simple optimization reduced the runtime approximately by a factor of 2 (13h 52m vs. 27h 11m).

## 4 Parallel Implementation

The tree-like organization of the branch-and-bound search space provides a potential for the parallelization of our algorithm, as all branches of the tree can be processed simultaneously. In this paper, we use two approaches to parallelize the algorithm: a shared-memory approach and a distributed-memory approach.

### 4.1 Shared-Memory Approach

In the shared-memory approach, all nodes $N_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,k}\}$, $i = \overline{1, I}$, $k = \overline{1, K_i}$, $K_i = \prod_{l=1}^{i} J_l$ at each layer $i$ of the tree are regarded as independent *tasks* that can be executed in parallel. The total number of tasks, $N_{tasks} = \sum_{i=1}^{G} K_i$, can be a very large number. Therefore, a *granularity* parameter $G$ is introduced to limit the degree of parallelism to a certain level of the tree: subtrees below the granularity level are not split into tasks but rather processed sequentially.

A pseudo-code for this approach is given in Listing 3. The main difference as compared to the sequential version is that recursive function calls are performed by newly created concurrent tasks (line 12). Besides, a copy of the current beginning part of the CES, `W`, has to be provided to each task.

The merit of this approach is its simple implementation: no communication is needed between tasks as they rely on shared memory for data exchange.

### 4.2 Distributed-Memory Approach

We use the master-worker paradigm for an alternative, distributed-memory parallelization of our algorithm: a single *master* process dispatches a subset of computations to multiple *worker* processes and gathers computed results from them.

**Master Process.** The master (see Listing 4) performs a depth-first traversal of the tree using a recursive procedure `MasterEnumerateVariants` to some level $G$ (*granularity*), $1 \leq G \leq I$. Using this procedure, the master creates beginning parts $W[i], i = \overline{1, G}$ of the CES (lines 24–29). At the last level of recursion, the master waits for worker messages (line 32), which can be of two types: *solution* (SOLUTION) or *job request* (REQUEST_WORK). If the master receives a *solution* message (line 33), the costs of the received solution are compared to the costs of the current optimal solution (optimality criterion, line 35). If a better solution has been found by the worker, it is stored and replaces the current optimal solution (lines 36–38). When a job request is received (line 39), the master responds by sending *job message* (DO_WORK) containing the current beginning part of the CES and the current optimal solution to the worker (line 40). Afterwards, a new beginning part of the CES is generated to be passed to a worker

```
1  FindSolution() { EnumerateVariants(0); }
2
3  EnumerateVariants(level) {
4    if (level < I) {
5      for (j=1; j <= J[level]; j++) {
6        /* append device variant to beginning part */
7        W[level] = X[level, j];
8        /* check compatibility constraint and upper bound */
9        if (S(W) == 0 && PartCost(W, level) < minCost) {
10         if (level < G) { /* check granularity */
11           /* create concurrent task */
12           CREATE TASK: EnumerateVariants(level + 1);  }
13         else {
14           /* search recursively */
15           EnumerateVariants(level + 1);   }   }   }   }
16   else { /* leaf node */
17     ... }
18 }
```

**Listing 3.** The shared-memory approach for parallel branch-and-bound.

(lines 25–29). If no new beginning part of the CES can be generated, the master returns from the recursive procedure `MasterEnumerateVariants` (line 6). The master continues receiving solutions from workers and compares them to the optimal solution. However, if a worker sends a job request, the master sends a *quit* message (QUIT) to the worker, to terminate the worker process. After quit messages have been sent to all workers, the master process ends.

**Worker Process.** The worker (see Listing 5) starts by sending a job request to the master (line 3) and waits for the response. The response can be of one of two types: *job message* (DO_WORK) or *quit* (QUIT). If a job message comprising a beginning part of the CES and the current upper bound of the optimality criterion is received, the worker calls the recursive procedure `WorkerEnumerateVariants` (line 5–8). Within this procedure, the worker traverses the remaining sub-tree $W[i], i = \overline{G+1, I}$ of the received CES' beginning part to find solutions in the same way the sequential algorithm does (lines 5–20 of Listing 1). If the worker finds a solution which costs do not exceed the upper bound of the optimality criterion (lines 24–27), it makes this solution the new optimal solution (lines 28–31). When the recursive procedure ends, the worker sends its new optimal solution, if any, to the master (line 9–11) and requests a new job. If a quit message is received, the worker process terminates (line 8–9).

The distributed-memory approach is more difficult to implement than the shared-memory approach: master-worker communication has to be specified explicitly in order to exchange data in a distributed-memory system. Besides, a single master constitutes a possible performance bottleneck of this implementation.

## 5 Experimental Results

To study the speedup of our two parallelization approaches, we created two corresponding implementations and conducted runtime experiments on a heterogeneous cluster consisting of:

```
1  Master() {
2    /* number of workers (one of processes is master) */
3    num_workers = NUM_PROCESSORS - 1;
4
5    /* start tree traversal */
6    MasterEnumerateVariants(0);
7
8    /* wait for remaining solutions and stop workers */
9    while (num_workers > 0) {
10     msg = ReceiveWorkerMessage();
11     if     (msg.type == SOLUTION) {
12       /* check optimality criterion */
13       if (Cost(msg.W) < minCost) {
14         /* make solution new optimal solution */
15         Wopt = msg.W;
16         minCost = Cost(msg.W);  }   }
17     elseif (msg.type == REQUEST_WORK) {
18       /* stop worker */
19       SendWorkerMessage(msg.workerID, QUIT);
20       num_workers--;  }   }
21 }
22
23 MasterEnumerateVariants(level) {
24   if (level < G) { /* check granularity */
25     for (j=1; j <= J[level]; j++) {
26       W[level] = X[level, j];
27       if (S(W) == 0 && PartCost(W,level) < minCost) {
28         /* search recursively */
29         MasterEnumerateVariants(level + 1); }   }   }
30   else {
31     while (true) {
32       msg = ReceiveWorkerMessage();
33       if     (msg.type == SOLUTION) {
34          /* check optimality criterion */
35          if (Cost(msg.W) < minCost) {
36            /* make solution new optimal solution */
37            Wopt = msg.W;
38            minCost = Cost(msg.W);  }   }
39        elseif (msg.type == REQUEST_WORK) {
40          SendWorkerMessage(msg.workerID, W, minCost);
41          break;  }   }   }
42 }
```

**Listing 4.** Distributed-memory approach: Pseudo-code of master.

- 36 nodes with 2 quad-core processors (Intel X5550 Nehalem, running at 2.6 GHz) with 3 GB RAM each,
- 198 nodes with 2 hexa-core processors (Intel Westmere X5650, running at 2.6 GHz) with 2 or 4 GB RAM each, and
- 4 nodes with 4 eight-core processors (Intel Xeon E7550, running at 2 GHz) with 128 GB RAM each.

The nodes are interconnected via Infiniband. Programs were compiled using the Intel C++ Compiler 11.1.

We study the design of a CES consisting of 16 processing stages with 5 variants of devices at every stage as test case. The implementations are written in C++ using OpenMP version 3.0 (Open Multi-Processing) [14] for the task-based approach (see Section 4.1), and the Message Passing Interface (MPI) [12] for the master-worker approach (see Section 4.2).

The implementation using OpenMP is derived from the sequential implementation by inserting directives: a `parallel` construct with a nested `single` construct is put around the call of the `EnumerateVariants` (line 1 of Listing 3), such that one of these threads starts the recursive tree traversal, while the other threads stay idle. Within the recursive procedure new tasks are created using the `task` construct of OpenMP. While the first thread continues creating tasks, the other threads process these tasks.

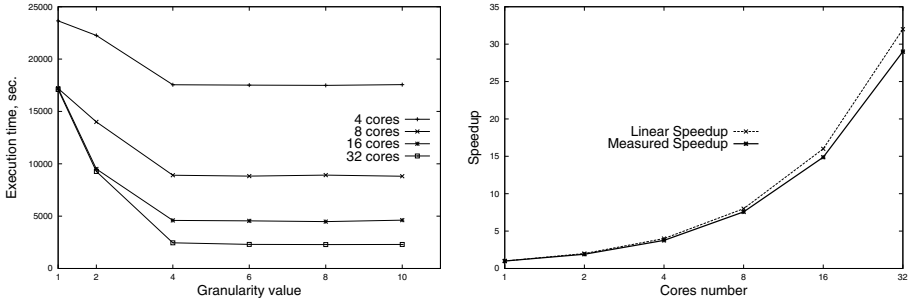
We run our OpenMP-based implementation on a single node consisting of 4 CPUs with altogether 32 cores, setting granularity values from 1 to 10. We observed that for granularity greater than 10, too many tasks were created, such that the implementation ran out of memory. The results are shown in Figure 2a. Figure 2b shows the speedup of our OpenMP-based implementation using up to 32 cores. Granularity has been set to 10.

In our master-worker implementation, we use MPI's point-to-point communication functions `send` and `recv` for exchanging messages between master and worker. We performed the same measurements on up to 64 Westmere nodes. Here, we also observed best performance for granularity values from 4 to 14 (see Figure 3a). The minimum number of processors for running the program is two (master and one worker). While there is no speedup when using 2 processors, it increases nearly linearly when using up to 768 processors. With greater numbers of processors, the growth of speedup slows down. The performance of the master process may become a bottleneck of application performance when it controls too many worker processes, because the master frequently communicates with all workers.

Both implementations provide high scalability. On the same hardware, the performance of both approaches differs slightly. However, in spite of its more difficult implementation, the MPI implementation is preferable, because it runs both on shared-memory machines and on computers with distributed memory. Currently, shared-memory machines with more processors (hundreds and thousands) are rare, unlike computing clusters. Also with a large number of tasks we may not have enough memory as in our case when $G > 10$.

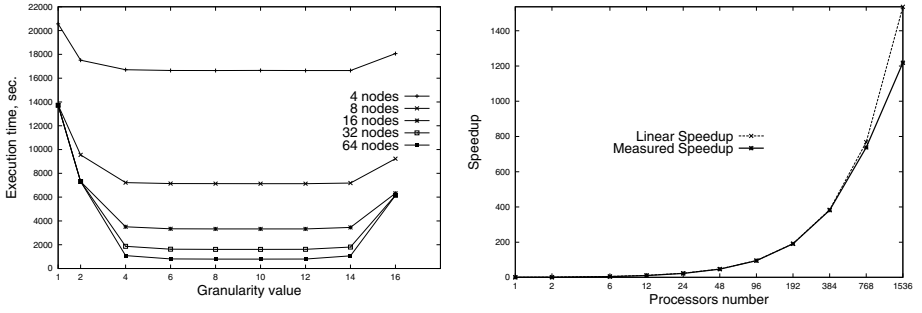
```
1  Worker() {
2     while (true) {
3       SendMasterMessage(workerID, REQUEST_WORK);
4       msg = ReceiveMasterMessage();
5       if (msg.type == DO_WORK) {
6         minCost = msg.minCost;
7         foundNewSolution = false;
8         WorkerEnumerateVariants(G + 1);
9         if (foundNewSolution) {
10          /* send new optimal solution to master */
11          SendMasterMessage(workerID, SOLUTION, Wopt);    }   }
12      elseif (msg.type == QUIT) {
13        break;  }   }
14 }
15
16 WorkerEnumerateVariants(level) {
17   if (level < I) {
18     for (j=1; j <= J[level]; j++) {
19        /* append device variant to beginning part */
20        W[level] = X[level, j];
21        /* check compatibility constraint and upper bound */
22        if (S(W) == 0 && PartCost(W, level) < minCost) {
23          /* search recursively */
24          WorkerEnumerateVariants(level + 1); }   }   }
25   else { /* leaf node */
26     /* check optimality criterion */
27     if (Cost(W) < minCost)) {
28       /* check processing time constraint */
29       if (T(W) <= Tmax) {
30         /* make solution new (local) optimal solution */
31         Wopt = W;
32         minCost = Cost(Wopt);
33         foundNewSolution = true;    }   }   }
34 }
```

**Listing 5.** Distributed-memory approach: Pseudo-code of worker.




(a) Runtime depending on granularity. (b) Speedup.

**Fig. 2.** Experimental results for the OpenMP-based implementation




(a) Runtime depending on granularity. (b) Speedup.

**Fig. 3.** Experimental results for the MPI-based implementation

Selecting a suitable granularity value is crucial for optimal performance. Usually, granularity should be set to a value, such that the number of initial parts for a system is significantly greater than the number of processors, i. e. $\prod_{i=0}^{G} J_i \gg N_p$. However, the distribution of initial parts to processors may become unbalanced if initial parts for a systems are discarded early by the branch-and-bound paradigm. Hence, we empirically determined a factor to optimize load balance. For the above example (16 processing stages with 5 device variants at each), this factor is 2–3, such that a sensible granularity value $G$ is within $2 \cdot \log_5 N_p \leq G \leq 3 \cdot \log_5 N_p$.

## 6 Conclusion

We proposed two approaches to implement a parallel branch-and-bound algorithm for solving the optimization problem for multi-product batch plants.

Implementations of our approaches based on OpenMP and MPI have been presented. Runtime experiments for our implementations using a real-world example of a multi-product batch plant show that our solution provides considerable speedup. This is well correlated with experimental results obtained in, e. g., [6], where also near-linear speedups were observed. Both implementations provide good parallel scalability.

We also analyzed the impact of the degree of parallelism controlled by a granularity parameter. From our results we conclude that while the MPI-based implementation suffers a communication bottleneck for large numbers of processors (the reasons for that and methods of overcoming are described in detail in [1]), it still provides better performance and flexibility as compared to the OpenMP-based implementation.

In future work we will investigate the use of a hierarchical master-worker implementation, in order to reduce the communication bottleneck which we observed in our current implementation. This paper presents a parallel version only of the branch-and-bound algorithm. In addition, quite interesting would be the parallelization of comprehensive mathematical model of CES operation. This problem requires also deeper and more detailed research in further works.

**Acknowledgement.** This work was supported by the DAAD (German Academic Exchange Service) and by the Ministry of Education and Science of the Russian Federation under "Mikhail Lomonosov II"-Programme.

## References

1. Aida, K., Natsume, W., Futakata, Y.: Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In: Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003), pp. 156–164 (2003)
2. Bouziane, H.L., Pérez, C., Priol, T.: Extending software component models with the master-worker paradigm. Parallel Computing 36(2-3), 86–103 (2010)
3. Brassard, G., Bratley, P.: Fundamentals of Algorithmics. Prentice-Hall, Englewood Cliffs (1996)
4. Cauley, S., Balakrishnan, V., Hu, Y.C., Koh, C.K.: A parallel branch-and-cut approach for detailed placement. ACM Transactions on Design Automation of Electronic Systems (TODAES) 16(2), 18:1–18:19 (2011)
5. El Hamzaoui, Y., Hernandez, J., Cruz-Chavez, M., Bassam, A.: Search for Optimal Design of Multiproduct Batch Plants under Uncertain Demand using Gaussian Process Modeling Solved by Heuristics Methods. Berkeley Electronic Press (2010)
6. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: Survey and synthesis. Operations Research 42(6), 1042–1066 (1994)
7. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, Design and Analysis of Algorithms, 2nd edn. Addison-Wesley, Reading (2003)
8. Leyffer, S., Linderoth, J., Luedtke, J., Miller, A., Munson, T.: Applications and algorithms for mixed integer nonlinear programming. Journal of Physics: Conference Series 180(1), 12–14 (2009)

9. Malygin, E., Karpushkin, S., Borisenko, A.: A mathematical model of the functioning of multiproduct chemical engineering systems. Theoretical Foundations of Chemical Engineering 39(4), 429–439 (2005)
10. Mansa, B., Roucairol, C.: Performances of parallel branch and bound algorithms with best-first search. Discrete Applied Mathematics 66(1), 57–74 (1996)
11. Martí, R., Gallego, M., Duarte, A.: A branch and bound algorithm for the maximum diversity problem. European Journal of Operational Research 200(1), 36–44 (2010)
12. Message Passing Interface Forum: Message Passing Interface Standards Documents, http://www.mpi-forum.org
13. Moreno, M.S., Montagna, J.M.: Multiproduct batch plants design using linear process performance models. American Institute of Chemical Engineer Journal 57(1), 122–135 (2011)
14. OpenMP Architecture Review Board: The OpenMP API specification for parallel programming, http://www.openmp.org
15. Ponsich, A., Azzaro-Pantel, C., Domenech, S., Pibouleau, L.: Mixed-integer nonlinear programming optimization strategies for batch plant design problems. Industrial & Engineering Chemistry Research 46(3), 854–863 (2007)
16. Rasmussen, M., Stolpe, M.: Global optimization of discrete truss topology design problems using a parallel cut-and-branch method. Computers & Structures 86(13-14), 1527–1538 (2008)
17. Rebennack, S., Kallrath, J., Pardalos, P.M.: Optimal storage design for a multiproduct plant: A non-convex minlp formulation. Computers & Chemical Engineering 35(2), 255–271 (2011)
18. Reinefeld, A., Schnecke, V.: Work-load balancing in highly parallel depth-first search. In: Scalable High-Performance Computing Conference, pp. 773–780 (1994)
19. Sanders, P.: Better algorithms for parallel backtracking. In: Ferreira, A., Rolim, J.D.P. (eds.) IRREGULAR 1995. LNCS, vol. 980, pp. 333–347. Springer, Heidelberg (1995)
20. Wang, Z., Jia, X.P., Shi, L.: Optimization of multi-product batch plant design under uncertainty with environmental considerations. Clean Technologies and Environmental Policy 12, 273–282 (2010)
21. Zhou, J., Yu, K.M., Lin, C., Shih, K.C., Tang, C.: Balanced multi-process parallel algorithm for chemical compound inference with given path frequencies. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010. LNCS, vol. 6082, pp. 178–187. Springer, Heidelberg (2010)

# Virtual Path Implementation of Multi-stream Routing in Network on Chip

Bartosz Chojnacki, Tomasz Maka, and Piotr Dziurzanski

West Pomeranian University of Technology, Faculty of Computer Science and Information Technology, ul. Zolnierska 49, 71-210 Szczecin, Poland
{bchojnacki,tmaka,pdziurzanski}@wi.zut.edu.pl

**Abstract.** In this paper, a multi-path routing algorithm dedicated to Network on Chip (NoC) together with its implementation are presented. The proposed algorithm is based on the Ford-Fulkerson method and is aimed at data-dominated streaming multimedia applications realized in Multi Processor Systems on Chip. The efficiency of the proposed technique is compared with the state-of-the-art NoC routing approach and in some cases we obtain a significant improvement. Our implementation utilizing virtual channels, despite imposing some overhead, allow us to obtain a promising results in some popular multimedia codecs.

**Keywords:** Network on Chip, multi-path routing, Ford-Fulkerson method.

## 1 Introduction

Contemporary multimedia algorithms and many others, are typically computational-intensive and data-dominated but they can be split into stages to be implemented in separate computational units. Thanks to this property they can benefit from parallel and distributed processing working in a pipeline-like way and transmitting each other streams of relatively large, but usually fixed, amount of data. In these applications, it is usually required to keep an assumed quality level of service and meet real-time constraints [11]. Multi Processor Systems on Chips (MPSoCs) are often considered as suitable hardware implementations of these applications [7]. As each processing unit of a MPSoC can realize a single stage of streaming application processing, it is still problematic to connect these units together.

They are often linked using the packet-based Network-on-Chip (NoC) paradigm for designs of chips realizing distributed computation [2]. The recent popularity of this approach can be attributed to a lower number of conflicts in a chip with a large number of cores. It is reported that NoC architectures offer high bandwidth and good concurrent communication capability, but they require additional mechanisms to overcome problems typical for packet switching communication, such as packet deadlock or starvation, but the techniques known for traditional computer networks have to be altered before applying to on-chip networks [10]. A typical NoC implementation utilizes packet switching approach that is called wormhole routing [9]. In this technique, each packet is split into

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 431–436, 2011.

smaller units of equal length, flits (flow control digits). Usually, the first flits contain some routing information, such as the destination address. Having obtained the routing information, a wormhole router selects the next-hop router and establishes the path to that neighbouring router. This path can be used exclusively for transferring the current package flit by flit as long as the whole package has not been transferred. The next-hop router typically does not store the whole package in its buffers, but tries to establish a connection with another router being selected for the transfer. The most popular routing algorithm used in NoCs, named XY, can be also viewed as inappropriate for switching large streams of information. According to this algorithm, a flit is firstly routed according to the X axis as long as the X coordinate is not equal to the X coordinate of the destination core, and then the flit is routed vertically. However, as it was shown in [5], in a mesh-based NoC realizing a typical streaming multimedia algorithm, few links are used significantly while the remaining ones are utilized in a small degree and relatively large part of links are not utilized at all. Taking into consideration the above mentioned facts, it follows that in order to design a NoC-based MPSoC for multimedia streaming applications it is necessary to propose a routing algorithm that is more suitable to this task that the traditional XY algorithm and to propose a mapping scheme of IP cores into mesh nodes that decreases the contention level. In this paper, we focus on the first of these issues.

## 2 Tapeworm Routing

In order to avoid the majority of problems found in a usage of the XY algorithm for streaming multimedia applications, we introduced a multi-path routing scheme that we named Tapeworm routing. We propose this name due to the similarity with the anatomy of a tapeworm - both its body and a package body are split into segments; segments are comprised of a number of flits.

The Tapeworm algorithm uses the well-known Ford-Fulkerson [6] method to compute maximal throughput of the network between a set of cores and for each core permutation. The Ford-Fulkerson method for an arbitrary flow network $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of edges with source $s$ and sink $t$. Each edge has capacity $c(u, v)$ and flow $f(u, v)$, defines the notion of the residual network and augmenting path $(u, v)$. The residual network for flow network $G$ is network $G_f = (V, E_f)$, where $E_f$ is defined as follows: $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$, where $c_f(u, v)$ denotes the residual capacity for path $(u, v)$, which is defined with $c_f(u, v) = c(u, v) - f(u, v)$. The augmenting path for a network is any path from $s$ to $t$ in a residual network for $G$. The residual capacity for any augmenting path for network is determined with the following formula: $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$.

With the notions defined as above, we can present the Ford-Fulkerson method in a pseudo-code given in Fig. 1. In case of the Tapeworm algorithm, the input data is a list of data transfers. A transfer $T_i$ is consisted of three elements $(S_i, D_i, A_i)$, where $S_i$ and $D_i$ denote the source and the target router, respectively, and $A_i$ is a number of bits transmitted between these routers. The

1. **while any augmenting path exists** $p \in G_f$
2. **for each** $(u, v) \in p$ **do**
3. $f(u, v) := f(u, v) + c_f(p)$
4. $f(v, u) := f(v, u) - c_f(p)$

**Fig. 1.** Ford-Fulkerson algorithm

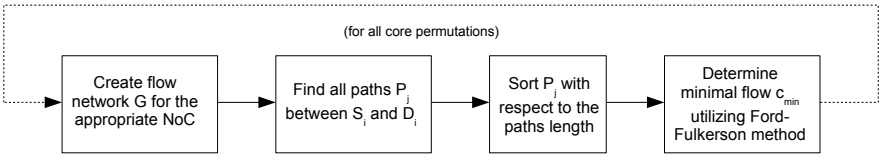



**Fig. 2.** Work flow of the transfer balancing process

pseudo-code of the whole Tapeworm algorithm is outlined in Fig. 2. Its details has been presented in [5].

Each router owns a routing table that stores all the paths to the target router sorted according to their lengths; paths of the same length are sorted with the XY rule. Following this rule, the first path is obtained based on the XY routing algorithm. In the second path, the flit is routed horizontally as long as the X coordinate is lower (or higher, according to the direction in X between the source and the target nodes) by 1 from the target router. Then the flit is routed vertically by one router, and then according to the XY algorithm. In the next path, a flit is routed horizontally as long as the X coordinate is lower (or higher) by 2 of the target router, etc. This approach guarantees receiving the flits in the same order as they were sent [4].

The Tapeworm and the XY algorithms include some common properties, as, for example, dealing with deadlocks by limiting the possibility of flit turning [1] and by utilizing the wormhole type of switching. The difference between these algorithms is clearly visible in the number of paths used to transmit data between two routers. A router is consisted of 4 functional blocks. Four buffers receive data from their neighbouring routers and one buffer receives data from the directly connected core. In the scheme, there is no output buffers as the Tapeworm algorithm operates according to the Wormhole switching technique, that permits to buffer flits of a single package in a few routers at once, such that the routed flits can be immediately transferred into input buffers of the next router. Data from buffers is then transferred to switch, which implements the Tapeworm algorithm. Moreover, switch controls the flit flow in router, reserving inputs for the corresponding outputs and works also as an arbiter.

In our approach we benefit from the virtual channel mechanism. The virtual channel is a pair of buffers in one physical channel which is shared with other such pairs. An access to a physical channel is controlled by a dedicated arbiter. Its introduction is aimed at getting rid of the deadlocks. This mechanism can

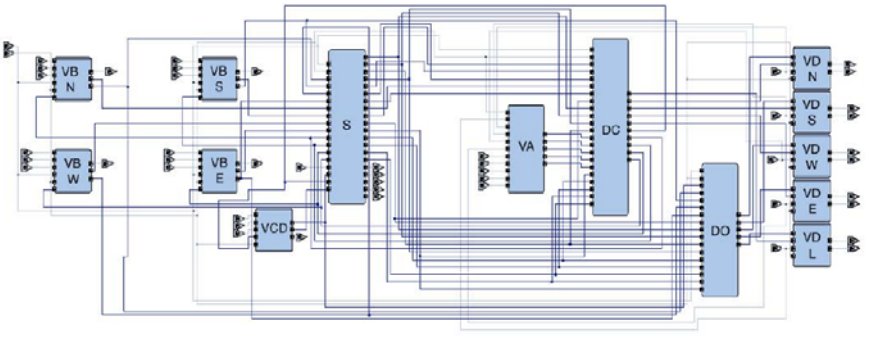



**Fig. 3.** Block diagram of the Tapeworm router

be also used for decreasing the delay in a network and for increasing its capacity [3]. Using virtual channels in the presented router is indispensable for implementing the Tapeworm algorithm. Tapeworm utilizes the Ford-Fulkerson method [6] to determine paths that can have common edges for various messages and thus can send flits originated from various messages using the same channels in the same time. In the described router architecture, it is possible to realize up to 3 virtual channels for each physical channel, which was enough for the analysed algorithm. However, this approach is scalable and the maximum number of virtual channels can be easily increased. The switch block is the most complex router block considering its implementation that consists of executing Tapeworm, which computes paths for transferring messages between the source and the destination router. Based on these paths, it creates the routing table for sending messages.

## 3 Experimental Results

The router architecture has been developed and implemented in CoCentric System Studio - a design and simulation environment allowing us to use the SystemC language. It is a C++ library aiming at designing of digital circuits that was developed in order to facilitate production of increasingly more complex systems consisting of a number of components, including the software ones. SystemC is capable of providing management of such systems designed at various levels of abstractions, what was impossible in case of the traditional hardware description languages. An architecture of the Tapeworm router is presented in Fig. 3, where VB denotes VirtualBuffer, VCD - VirtualCoreDemux, VD - VirtualData, S - Switch, VA - VirtualAck, DO - DataOut, DC -DataControl. In order to compare both the Tapeworm and XY routing algorithms, we performed one default mapping of functionalities into cores.

In Fig. 4a and 4b one can observe that the maximal time needed for sending and receiving data through the network generating traffic for the H.264 codec

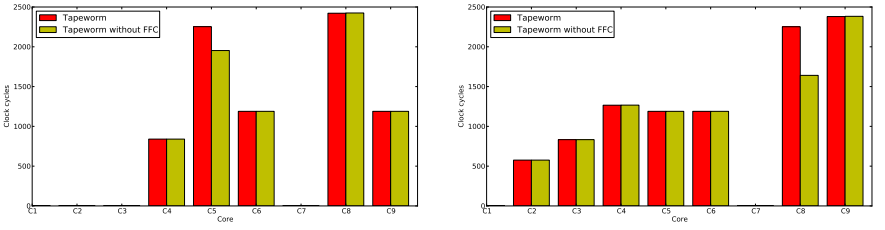



**Fig. 4.** Receiving (a) and sending (b) times for H.264 codec

and utilizing the Tapeworm routing scheme is lower than in the XY routing algorithm case by over 19%. In case of the Tapeworm algorithm, the standard deviation of sending and receiving flits is lower by 12% and 13%, respectively.

## 4 Conclusion

In this paper, we showed an architecture of a router implementing the Tapeworm routing algorithm realized in the SystemC language. Then some survey on the efficiency of a Network on Chip using the implemented router for three different applications have been shown. The obtained results have been compared with the network realizing the same applications using the XY algorithm. The Tapeworm algorithm was meant to result in faster networks with more balanced transfers [4] [5], and, in case of the H.264 video encoder, the Tapeworm approach is faster by more than 19%. The Tapeworm algorithm requires further research, for example with different implementations of the virtual channels. Additionally, it can be interesting to perform the tests with various message size, bit-width of the links or input buffers in routers, that have not been presented in the paper.

The research work presented in this paper was sponsored by Polish Ministry of Science and Higher Education (years 2011-2014).

## References

1. Bjerregaard, T., Mahadevan, S.: A Survey of Research and Practices of Network-on-Chip. ACM Computing Surveys (CSUR) 38, Article 1 (2006)
2. Dally, W.J., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann Publisher, San Francisco (2005)
3. Duato, J., Yalamanchili, S., Ni, L.: Interconnection Networks. An Engineering Approach. Morgan Kaufmann Publishers, San Francisco (2003)
4. Dziurzanski, P., Maka, T.: Stream-based Multi-path Routing Scheme in On-chip Networks. In: 16th EUROMICRO Conference on Parallel, Distributed and Network-based Processing PDP, Toulouse, France, February 13-15, pp. 15–16 (2008)
5. Dziurzanski, P., Maka, T.: Stream transfer balancing scheme utilizing multi-path routing in networks on chip. In: Woods, R., Compton, K., Bouganis, C., Diniz, P.C. (eds.) ARC 2008. LNCS, vol. 4943, pp. 294–299. Springer, Heidelberg (2008)

6. Ford Jr., L.R., Fulkerson, D.R.: Flows in Networks. Princeton University Press, Princeton (1962)
7. Kavaldjiev, N., et al.: Routing of guaranteed throughput traffic in a network-on-chip. Technical Report TR CTIT-05-42 Centre for Telematics and Information Technology, University of Twente, Enschede (2005)
8. Lee, H.G., Chang, N., Ogras, U.Y., Marculescu, R.: On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. ACM Transactions on Design Automation of Electronic Systems (TODAES) archive 12(3) (2007)
9. Li, M., Zeng, Q.A., Jone, W.B.: DyXY: a proximity congestion-aware deadlock-free dynamic routing method for network on chip. In: 43rd ACM IEEE Design Automation Conference (DAC), pp. 849–852 (2006)
10. Ogras, U.Y., Marculescu, R.: Prediction-based Flow Control for Network-on-Chip Traffic. In: 43rd ACM IEEE Design Automation Conference (DAC), pp. 839–844 (2006)
11. Smit, G.J.M., et al.: Efficient Architectures for Streaming DSP Applications, Dynamically Reconfigurable Architectures. In: Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI). Schloss Dagstuhl, Germany (2006)

# Web Service of Access to Computing Resources of BOINC Based Desktop Grid

Evgeny Ivashko and Natalia Nikitina

Karelian Research Centre of the Russian Academy of Sciences
Institute of Applied Mathematical Research
{ivashko,nikitina}@krc.karelia.ru

**Abstract.** In the paper we describe a web service of access to computing resources of a desktop grid built in the High-performance Data Center of Karelian Research Centre of the RAS. The grid is based on lightweight, highly scalable BOINC platform. We present the architecture of the system, the current results of implementation and plans for the future. We also present the overview of a scientific research that was carried out with use of the system.

**Keywords:** distributed computing, volunteer computing, grid, BOINC.

## 1 Introduction

Recently, supercomputers have been supporting a lot of scientific applications demanding significant computing power. The most common examples include mathematical modeling, numerical computation, physical and biological simulation etc. But not only are supercomputers powerful in solving diverse scientific problems, they are also extremely expensive in use, maintenance and scaling. For applications that process huge amounts of data and at the same time feature a high degree of data parallelism, grid technology has become a priority. Grid computing has been applied by the National Science Foundation's [1] National Technology Grid, NASA's [2] Information Power Grid, European DataGrid [3], Russian Data Intensive Grid [4] etc.

By definition, a computational grid is "a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities" concerned with "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations" [7,8]. Grids allow to integrate computing power of a set of distributed computers connected by the network.

Grid systems can be divided into two groups: the first one includes grids that combine specially dedicated computers (usually supercomputers), the second one includes those utilizing resources of desktops while they are idle. A grid system combining personal computers connected to the Internet and voluntarily provided by their owners is called a system of volunteer computing [10]. A desktop grid [11] is a system that integrates resources of computers in a local network of an organization.

In February 2009, a computing cluster with peak performance of 850 GFlops was launched at the Institution of the Russian Academy of Sciences, Karelian Research

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 437–443, 2011.

Centre [12]. In order to provide access to computing resources for users and organize regular work of the cluster, the Center for collective use, "High-performance Data Center", was established on basis of Institute of Applied Mathematical Research (IAMR). Computing resources of the cluster are being used for solving computationally intensive tasks within scientific and applied research as well as for training of parallel programming specialists.

However, recently computing resources of the cluster have ceased to suffice for convenient work of users as the workload increased to 83%. As a rule, applications have to wait in queue until the necessary number of computational nodes is available. At the same time, it has been found that a whole number of users solve computational tasks that are well data-parallelized. Consequently, there was made a decision to create a grid within the framework of the Center for collective use.

Our objective is to implement a service of distributed execution of applications which require significant computational resources, are data parallel and can relatively easily be separated into a number of tasks. We intend to do this on basis of the desktop grid built on the BOINC software. The nodes of the grid are the cluster nodes, servers and desktop PCs of IAMR that will devote to the project their idle time.

## 2 Software Platform

There are a number of software systems, or middlewares, for grid computing. The most popular ones are Globus Toolkit [13], Unicore [14], GLite [15], Condor [16], ARC [17]. However, for the implementation of the web service we have chosen the BOINC platform [9], which was designed specially for volunteer computing and desktop grids. BOINC (Berkeley Open Infrastructure for Network Computing) is a free, distributed under the GNU LGPL license software platform for distributed computing, developed at the University of California, Berkeley. BOINC platform has been a framework for many independent volunteer computing projects [18-20]. In comparison with other popular middlewares that also support volunteer computing, such as Condor, BOINC is much more lightweight, less complicated in deployment and expansion, and available for a larger variety of platforms.

There have been several other systems for distributed computing with use of idle computers in a network**,** e.g. QADPZ [6], Bayanihan [23] and Entropia [24]. However, the BOINC software is the most actively developed at the moment and supports the widest range of applications [25]. Being cheap in deployment and maintenance, BOINC server software has high performance and good opportunities in scalability. The client part is available for many computing platforms and is easy to deploy. BOINC is an open source software, has a lot of features and provides documented interfaces to many of its key components.

One of the basic concepts of BOINC software is a *project*. A project is an autonomous entity that does distributed computing. Projects are independent; each one has its own applications, database, web site and servers, and is not affected by the status of other projects. Each project is identified by the URL of its web site.

A BOINC project can include multiple *applications*, each of which consists of several programs for different computing platforms and a set of *workunits* and *results*.

A workunit is an independent computational task. A result, each associated with a workunit, describes an instance of a computational task, either *unstarted*, *in progress*, or *completed*. In some cases there may be several instances, or "replicas", of a given workunit. This is used to ensure that clients don't send back to server intentionally or unintentionally wrong results. The BOINC server creates one or more instances of a computational task according to project preferences, distributes them to client hosts, collects the output files, handles the results and after all deletes the I/O files.

Typically, creation and support of a BOINC project involve the work of an administrator whose main duties are to manually create applications, fill in the preferences for computational tasks etc. Hence, to enable wider access to the computing resources of the desktop grid we have decided to implement a web service that is described further in this paper.

There are a few things that the term "web service" may refer to. We use a broad definition of the web service given by IBM [22]:

> *"A self-contained, modular application that can be described, published, located, and invoked over the Web. Platform-neutral and based on open standards, Web Services can be combined with each other in different ways to create business processes that enable you to interact with customers, employees, and suppliers."*

Each BOINC application includes the main executable (probably with other files necessary to run it) and descriptions of I/O files that are specific for this application and common for all its workunits. The administrator of the BOINC server must provide application-specific programs for validation and assimilation of the computational results. Given the main executable, descriptions of I/O files, validation and assimilation programs, a new application can be created and run automatically.

The Leiden Classical project [5] (which is originally aimed at supporting test simulations of molecules and atoms in a classical mechanics environment) has implemented a web-based interface allowing users to submit BOINC jobs, including uploading their own input files. However, the system allows to create workunits for registered BOINC applications only. Moreover, it lacks the feature of uploading many input files at a time and some other important functions such as custom number and format of I/O files. In our work the aim was to develop a web-based interface that would assist in creating users' applications that require different I/O parameters.

Any existing executable can be run under the BOINC software with use of a *wrapper program* supplied by BOINC [21], thereby without need of modification of the source code. The wrapper runs the executables, or "real" applications, as subprocesses and handles all communication with the core client. There may be more than one executable under the wrapper, e.g. a long task may be separated into multiple tasks for the purpose of enabling checkpointing. Both wrapper program and executable files must be built for the computing platforms that grid working nodes have. BOINC allows applications to have various versions for a wide range of computing platforms.

The client part in the desktop grid includes files that are necessary to run the client program, bind it to an account and allow remote control and monitoring of client. Grid server is directly connected to the web server.

## 3 The System Architecture

The web service of access to computing resources of the desktop grid is based on the following components:

- *website* that serves as an access point to the service and provides a user interface;
- *grid server* (based on BOINC platform) that controls creation and distribution of computational tasks and handles the results;
- *grid working nodes* that perform computational tasks.

The system architecture is shown in Fig. 1. The workflow in the system is described as follows:

- the user uploads files and specifies parameters required to create computational tasks via the website;
- the uploaded data is processed by a server-side program that creates a new BOINC application and computational tasks;
- instances of the new application together with one or more computational tasks are sent to working nodes that have enough resources to execute them;
- the user monitors execution progress via the website;
- the server collects the results from the working nodes, processes them and sends to the user.

## 4 Implementation

In Fig. 2 there is a screenshot of the web-based interface of the service. The form provides the fields to upload the executable, additional files (if any) and a zip archive of input files; the fields for physical names of I/O files that will the executable require; the fields for estimates of resources to complete the tasks and the field for email address which are the results to be sent to. The field for upload of the main executable and the field for email address are compulsory, others are optional.

A user enters the website, uploads the main executable and probably additional files, specifies the names for I/O files that the executable will require, and uploads a couple of input files, each one for a separate computational task. All uploaded data is processed by a server-side program that creates a new application and makes templates for input and output files. In our case every application contains the wrapper program in addition to user's executable. Computational tasks, or workunits in terms of BOINC, are generated with use of input files together with their templates and templates of output files that are to be received.

The grid working nodes periodically send to the server requests for more work. The BOINC scheduling program responds to work requests by distributing unsent results between the working nodes that have asked for work and meet certain criteria, e.g. have enough disk space and memory to handle the tasks.

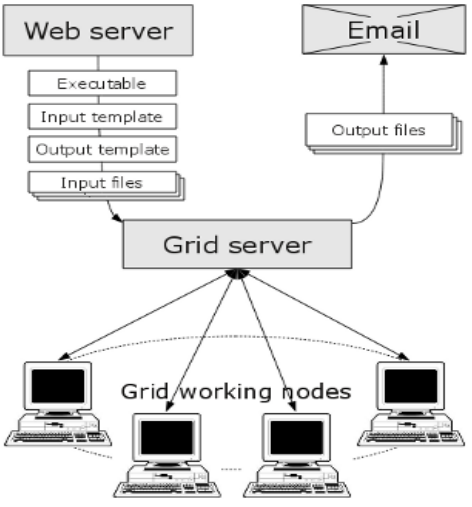



**Fig. 1.** The system architecture

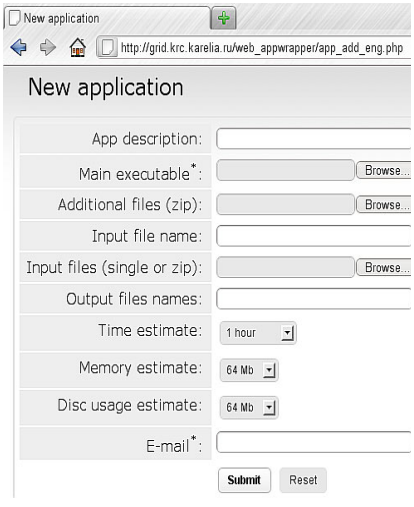



**Fig. 2.** Web interface of the service

The overall execution progress is displayed on the web site. At the moment the user may view the whole number of workunits and their status, i.e. in progress or completed. Once the completed result of a workunit is reported by a grid working node, a validation program checks the output files and marks the result either as valid or invalid. Valid results are then handled by an assimilation program that stores them in a zip archive and sends it to user via email as soon as all results have been handled.

Currently the number of users of the service is not restricted. There may be as many workunits as required and the time of their execution will depend only of the number of available nodes. However, BOINC allows to assign priorities to applications of different users.

## 5 Conclusion, Results and Further Work

In this paper we describe a web service of access to computing resources of a desktop grid based on the BOINC software and present the current results of its implementation. The service is intended mainly for applications that are data parallel and can relatively easily be separated into a number of tasks. The final version of the service will be available for users of the Center for collective use as well as users of other institutions within the agreements.

The web service has been used in solving the computational problem of finding the optimal parameters for the mathematical model of hydride decomposition within the scientific research devoted to studying the White Sea. This is a non-classical boundary-value problem with a free boundary and nonlinear Neumann boundary conditions. The finite-difference numerical method was implemented in Fortran-90/95. The problem is to find the set of kinetic parameters that provide the best (in the least squares sense) approximation of the desorption curve obtained experimentally. A difficulty is that the solution can be not unique.

Currently we have obtained the following results:

- the software platform has been selected;
- the system architecture has been designed;
- the BOINC based desktop grid has been deployed;
- sixteen working nodes have been added to the grid, among them ten cluster compute nodes, a cluster frontend, four servers and a desktop computer;
- the website has been created;
- a series of users' projects have been implemented.

Three main directions for the further work are:

- improvement of the web interface:
  - provide access to BOINC features such as automatic generation of workunits, validation of results, monitoring the whole progress of an application etc.;
  - implement personalization including management of personal accounts, viewing the statistics of use, access to the cluster, organization of the projects that may include several applications.
- grid development:
  - add more computers to serve as grid working nodes;
  - adjust parameters of server-side programs and application workunits so as to minimize waiting time;
  - implement a feature of uploading source code for further cross-compilation on the server side.
- scientific research:
  - develop a mathematical model of the job queueing process in the system and select the optimal queue discipline (e.g., the one that would guarantee the minimal average waiting time, or the minimal queue length, etc.);
  - investigate the statistical distribution of completion times of the workunits and develop a procedure of their adaptive assignment to those clients who will have enough resources with high probability;
  - investigate the dependence between size of the workunit and time for it to be completed in the grid system and develop a method to automatically separate the large problem into smaller workunits most effectively.

## References

1. The National Science Foundation, http://www.nsf.gov
2. The National Aeronautics and Space Administration, http://www.nasa.gov
3. European Data Grid Project, http://grid.web.cern.ch/grid
4. Russian Data Intensive Grid, http://www.egee-rdig.ru
5. The Leiden Classical project, http://boinc.berkeley.edu/trac/wiki/UserJobs
6. QADPZ - Quite Advanced Distributed Parallel Zystem, http://qadpz.sourceforge.net/

7. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
8. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications 15(3) (2001)
9. BOINC: open-source software for volunteer and grid computing, http://boinc.berkeley.edu
10. Volunteer computing, http://boinc.berkeley.edu/trac/wiki/VolunteerComputing
11. Grid computing with BOINC, http://boinc.berkeley.edu/trac/wiki/DesktopGrid
12. High-performance Data Center, KarRC, RAS, http://cluster.krc.karelia.ru
13. The Globus Toolkit, http://www.globus.org/toolkit
14. UNICORE: Distributed Computing and Data Resources, http://www.unicore.eu
15. Glite: Lightweight Middleware for Grid Computing, http://glite.web.cern.ch/glite
16. Condor: High Throughput Computing, http://www.cs.wisc.edu/condor
17. ARC: The Advanced Resource Connector, http://www.knowarc.eu
18. SETI@home, http://setiathome.berkeley.edu
19. Einstein@home, http://einstein.phys.uwm.edu
20. Rosetta@home, http://boinc.bakerlab.org/rosetta
21. The BOINC Wrapper, http://boinc.berkeley.edu/trac/wiki/WrapperApp
22. IBM(R) WebSphere(R) Host Publisher; Administrator's and User's Guide. Glossary, http://www.ibm.com/software/webservers/hostpublisher/library/publications/guide40/guide16.htm
23. Sarmenta, L.F.G.: Bayanihan: Web-Based Volunteer Computing Using Java (1998)
24. Chien, A., Calden, B., Elbert, S., Bhatia, K.: Entropia: Architecture and Performance of an Enterprise Desktop Grid System (2001)
25. Anderson, D., Christensen, C., Allen, B.: Designing a Runtime System for Volunteer Computing (2008)

# Solution of Assimilation Observation Data Problem for Shallow Water Equations for SMP-Nodes Cluster*

Evgeniya Karepova and Ekaterina Dementyeva

Institute of Computational Modelling of SB RAS,
660036 Akademgorodok, Krasnoyarsk, Russia
e.d.karepova@icm.krasn.ru
http://icm.krasn.ru
Siberia Federal University, 660041 pr. Svobodny, 79, Krasnoyarsk, Russia
http://www.sfu-kras.ru

**Abstract.** The ill-posed inverse problem of propagation of long waves in a domain of arbitrary form with the sufficiently smooth boundary on a sphere is consider. Numerical solution is based on finite elements method. Parallel software using MPI is performed. We compared efficiency of two popular implementations of the MPI standard and studied the behavior of our software when using various ways of memory allocation.

**Keywords:** Data assimilation problem, finite elements method and high performance computation.

## 1 Introduction

In this paper some problems of the effective use of cluster systems are considered, as an example paralleling implementation of the finite elements method for the numerical solution of an initial boundary value problem for the shallow water equations.

Shallow water models adequately describe a large class of natural phenomena such as large-scale free surface waves arising in seas and oceans, tsunamis, flood currents, surface and channel run-offs, gravitation oscillation of the ocean surface [1,2]. In the papers [2,3] the numerical modeling of free surface waves in large water areas on the basis of the shallow water equations (SWE) is considered taking into account the Earth's sphericity and the Coriolis acceleration. In [2] for the differential formulation of the problem useful a priori estimates which provide stability as well as existence and uniqueness of a solution of the problem are obtained. In [3] for this problem the finite elements method (FEM) is constructed and corresponding a priori estimates are obtained. Besides, numerical results on special model grids and on non-structured grids for water areas of the Sea of Okhotsk and the World Ocean are presented. In [4] efficiency of two parallel

* This research was supported by RFBR grant 11-01-00224-a, Integration Project 26 of SB RAS and Federal Special-Purpose Programme GK 02.740.11.0621.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 444–451, 2011.

implementations of the numerical solution of a boundary value problem for SWE with the use of the MPI library for the C language is studied.

When developing parallel software for this problem, we faced some difficulties caused by lack of information of efficiency of a certain method for the solution of particular problem. In this context, we have performed a study whose results are related not only to the problem itself but to a greater extent to tools for its solution. In particular, we compared efficiency of two popular implementations of the MPI standard and studied the behavior of our software when using various ways of memory allocation.

## 2 The Differential Formulation of a Problem

We formulate the problem on propagation of long waves in a water area as follows. Let $\Omega_{R_E}$ be a given domain on a sphere of radius $R_E$ with the boundary $\Gamma$=$\Gamma_1 \cup \Gamma_2$ where $\Gamma_1$ is the part of boundary passing along the coast and $\Gamma_2 = \Gamma \setminus \Gamma_1$ is the part of boundary passing through the water area. Denote the characteristic function of these parts of the boundary by $m_1$ and $m_2$, respectively. Without loss of generality we can assume that the points $\varphi = 0$ and $\varphi = \pi$ (poles) are not involved in $\Omega_{R_E}$. Assume also that $\Omega = \{(\lambda, \varphi) \in [0, 2\pi] \times (0, \pi) : (R_E, \lambda, \varphi) \in \Omega_{R_E}\}$, where $\lambda$ denotes longitude, $\varphi$ – latitude. For the unknown functions $u = u(t, \lambda, \varphi)$, $v = v(t, \lambda, \varphi)$ and $\xi = \xi(t, \lambda, \varphi)$ in $\Omega_{R_E} \times (0, T)$ we write the impulse balance equation and the continuity equation [2]:

$$\begin{aligned} \frac{\partial u}{\partial t} &= lv + mg\,\frac{\partial \xi}{\partial \lambda} - R_f u + f_1, \\ \frac{\partial v}{\partial t} &= -lu + ng\,\frac{\partial \xi}{\partial \varphi} - R_f v + f_2, \\ \frac{\partial \xi}{\partial t} &= m\left(\frac{\partial}{\partial \lambda}(Hu) + \frac{\partial}{\partial \varphi}\left(\frac{n}{m} Hv\right)\right) + f_3, \end{aligned} \tag{1}$$

where $u$ and $v$ are components of the velocity vector $\mathbf{U}$ in $\lambda$ and $\varphi$ direction, respectively; $\xi$ is deviation of a free surface from the nonperturbed level; $H(\lambda, \varphi) > 0$ is the depth of a water area at a point $(\lambda, \varphi)$; the function $R_f = r_*|\mathbf{U}|/H$ takes into account base friction force, $r_*$ is the friction coefficient; $l = -2\omega \cos\varphi$ is the Coriolis parameter; $m = 1/(R_E \sin\varphi)$; $n = 1/R_E$; $g$ is gravitational acceleration; $f_1 = f_1(t, \lambda, \varphi)$, $f_2 = f_2(t, \lambda, \varphi)$ and $f_3 = f_3(t, \lambda, \varphi)$ are given functions of external forces.

We consider boundary conditions in the following form:

$$HU_n + \beta m_2 \sqrt{gH}\xi = m_2 \sqrt{gH} d \quad \text{on} \quad \Gamma \times (0, T), \tag{2}$$

where $U_n = \mathbf{U} \cdot \mathbf{n}$, $\mathbf{n} = (n_1, \frac{n}{m} n_2)$ is the vector of an outer normal to the boundary; $\beta \in [0, 1]$ is a given parameter, $d = d(t, \lambda, \varphi)$ is a function defined on the boundary $\Gamma_2$.

We also impose initial conditions

$$u(0, \lambda, \varphi) = u_0(\lambda, \varphi),\ v(0, \lambda, \varphi) = v_0(\lambda, \varphi),\ \xi(0, \lambda, \varphi) = \xi_0(\lambda, \varphi).$$

For time discretization we subdivide the segment $[0, T]$ into $K$ subintervals by points: $0 = t_0 < t_1 < \cdots < t_K = T$ with the step $\tau = T/K$. We approximate time derivatives with backward differences and consider the system (1)–(2) on each instance $(t_k, t_{k+1})$. Using linearization in friction term from previous time level we obtain the semi-discrete elliptic type system.

In the general case the function $d$ is unknown, therefore, to close the problem (1)–(2), we consider a closure equation defined on some part $\Gamma_0$ of the boundary with the characteristic function $m_0$:

$$m_0 \xi = \xi_{obs}, \tag{3}$$

where $\xi_{obs} \in L_2(\Gamma_0)$ is a given function (for example, from observation data). Thus, at each time step the differential problem can be formulated as the problem on observation data assimilation in the following way [2].

*Assume that $\xi_{obs}$ is defined on $\Gamma_0$, the function $d$ is unknown on $\Gamma_2$ and vanishes on $\Gamma_1$. Find $u$, $v$, $\xi$, $d$ satisfying the system* (1)*, the boundary condition* (2) *and the closure condition* (3) on each time instance $(t_k, t_{k+1})$.

To solve this ill-posed inverse problem an approach, based on optimal control methods and adjoint equations theory, is used. We consider two family of optimal control problem with regularization to determine of minimum of calculating error between $\xi^h$ numerical free surface level and $\xi_{obs}$ observation one with respect to some special norm. The iterative numerical method to recovery of the boundary function (and, hence, to obtaining a general solution of our problem) is suggested. This method consists in iterative improvement of the boundary function by numerical solution of direct and adjoint problems by turns.

Numerical solution of direct and adjoint problems is based on finite elements method.

Consider a consistent triangulation $\mathcal{T} = \{\omega_i\}\big|_{i=1}^{N_{el}}$ of the domain $\Omega$ [5]. The Bubnov-Galerkin method is used for discretization of our problem with respect to space. Linear functions on triangular finite elements are used as trial and test functions. As a result, we obtain large dimension system of linear algebraic equations. A priori stable estimations are derived for discrete analogue [3]. The second order of approximation in internal nodes was shown.

## 3 Parallel Implementation. An Estimate of Potential Acceleration of a Parallel Algorithm

When solving the system of linear algebraic equation, the Jacobi iterative process is used. It can be effectively parallelized, besides, diagonal dominance for its convergence is easily provided by the choice of time step $\tau$.

Notice some features of implementation of the algorithm related to the finite element method. The global stiffness matrix depends on time and it must be recalculated at each time step. However, when implementing the Jacobi method on finite elements, there is no need to store a global stiffness matrix explicitly. In the program only elements of local stiffness matrices are calculated, moreover, only their diagonal elements depend on time and are recalculated at each

time step. Residual is assembled over triangles with the use of elements of local stiffness matrices.

Basing on explicitly parallelizing with respect to data, we can subdivide an original computational domain into several partially overlapping subdomains. In each subdomain calculation are performed independently of each other in the framework of a Jacobi iteration step. After each Jacobi iteration step, data adjustment in overlapping is required. We use the decomposition without shadow lines. An original domain is subdivided into subdomains which intersect each other only in boundaries of a cut. For each boundary point of a subdomain, residual is partially calculated only over the triangles lying in this subdomain. When exchanging data after each Jacobi iteration step, additional summation for the values of residual at boundary points of a subdomain is required.

The parallel program is implemented in the C language with the use of functions of the MPI library.

In the context of the data distribution scheme, all processes perform the same calculations but over different subdomains. The exchange structure is homogeneous except the first and the last processes. After each Jacobi iteration step, a process exchanges data with all its neighbours, the number of neigbours is defined by decomposition and does not depend on the number of processes involved in calculations.

Potential acceleration of algorithm is estimated as the ratio of the time $T_1$ of calculation with one processor to time $T_p$ of calculation with $p$ processors: $S_p = T_1/T_p$. We make theoretical estimates of potential acceleration, taking into account the time required for exchanges as well as the cost of calculations in overlapping subdomains wherever possible.

We denote the time of performing one arithmetic operation by $t_{op}$ and the time of performing transfer of one value by $t_{comm}$. It is clear that more likely the latter quantity is a virtual property of transfer rate, however, it is suitable for theoretical estimates.

Assume that $N_{nd}$ is the total number of nodes of a computational domain, $s$ is the number of operations performed at one node at each Jacobi iteration step, $k$ is the number of time steps, $\nu$ is the average number of Jacobi iteration steps at each time step. Then the time of performing the algorithm with one processor can be estimated as follows: $T_1 \sim k\nu s N_{nd} t_{op}$.

Assume that, when decomposing a domain, we can distribute uniformly all calculations among processors. In this case for the time of performing the algorithm with $p$ processors we can write the following:

$$T_p \sim T_1/p + T_{over} + T_{comm}. \tag{4}$$

Here $T_{over}$ is the time for additional calculations related to decomposition of a domain, $T_{comm}$ is the time for exchanges.

From the data distribution scheme it follows that at each Jacobi iteration step the following operations must be performed: 1) a global reduction operation for calculation of the halt criterion for the iterative process $T^1_{comm} \sim (t_{op} + t_{comm}) \log_2 p$; 2) exchange of values of a part of a residual vector at each point

of a cut $T^2_{comm} = \mu k \nu m N_{bnd} t_{comm}$ where $m$ is the number of values which must be transferred to a neighbouring process for one point of a cut, $\mu$ is the number of neighbouring processes involved in exchange; 3) additional summation of parts of residual for three components of the vector $(u, v, \xi)$ calculated with a neighbouring processor $T_{over} \sim k \nu g N_{bnd} t_{op}$, , hence, $g = 3$.

As the result, the estimate for potential acceleration of the parallel algorithm in the case of nonblocking two-point exchanges has the form:

$$S_p = \frac{1}{\frac{1}{p} + \frac{g}{s}R + \frac{\log_2 p}{sN_{nd}}(1+\kappa) + \frac{\mu m}{s}R\kappa}. \tag{5}$$

From the estimate (5) it follows that for relatively fine grids potential acceleration is close to linear one for sufficiently large range of the number of processes. The magnitude of acceleration is defined by two parameters. The first parameter $R = N_{bnd}/N_{nd}$ characterizes decomposition of a computational domain. Reasonable acceleration is provided by a small value of $R$, hence, when constructing decomposition of a complicated computational domain, along with the requirement of equal computational load per a process, it is, necessary to provide minimal length of a boundary of a subdomain for each process. The second parameter $\kappa = t_{comm}/t_{op}$ characterizes communication environment. This parameter describes a computational network rather arbitrary, but it shows that for reasonable acceleration one should choose architecture of a cluster with small values of $\kappa$. We notice once more that the magnitude of $T_{over}$ is less by several orders than other terms in the denominator in (4) and it does not depend on the number of processes. Taking into account memory saving and ease of implementation on a nonstructured grid, this provides the advantage of decomposition without overlapping.

## 4 Comparison of Two MPI Implementations and of Memory Management Strategies

Memory management strategy was analyzed with the cluster system of ICM SB RUS (in-house assembly, 96 cores, performance for the LINPACK test is 300 GFlops). Of course, results of this comparison, which is performed for a specific problem with a specific supercomputer, may not be generalized immediately to the case of arbitrary architecture of a cluster and software. However, they are rather interesting.

In this paper, performance of two popular MPI implementations (well-known MPICH2 v.1.2.1p1 and OpenMPI v.1.4.1 being a "heritor" of the LAM package) is compared.

Two modifications (with static and dynamic – *calloc* - *free* – memory allocation for main arrays and buffers) of the problem were tested. We immediately notice that the version with static allocation does not show considerable advantage of any package. The difference in running time and in time of data exchanges for all tested configurations turns out to be too small to be taken into account.

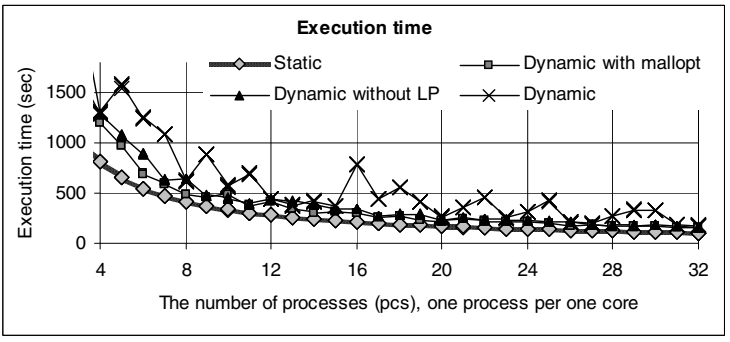



**Fig. 1.** Graphs of dependence of execution time on the number of nodes being used for different memory management strategies

Conversely, the version with dynamic memory allocation shows dependence on a package being used and on its settings. In the strict sense, the studies show that the differences in the behavior of the problem are related to distinctions in dynamic memory management rather then to features of implementation of MPI functions in both packages.

In particular, in the OpenMPI package the *ptmalloc* memory manager is used for dynamic memory allocation. It is applied for fragmentation control as well as for improving performance of an application due to acceleration of operation of the *malloc/free* procedure. A setting managing an allocation/disposal memory strategy is called *mpi_leave_pinned* and on default this setting is on. In MPICH2 package there is no such a setting, however, there is a possibility to manage the strategy which is used by the *glibc* system library when processing a memory allocation request by the call of the *mallopt()* function with corresponding arguments.

In Fig. 1, graphs of dependence of execution time of a program on the number of nodes for different strategies of memory management are shown. The best results in running time and agreement with the theoretical estimates (5), which are expressed in smoothness of a curve, are shown by the version of the program managing static memory.

The second result in running time and agreement with theory is shown by the "*dynamic allocation + mallopt*" strategy (the MPICH2 package) where memory is dynamically allocated but the command not to use the *mmap* mechanism (memory pages mapping) for memory allocation is given to the system library. In condition where an application uses OS Linux resources in the exclusive usage mode (allocates large memory space one time when starting and deallocates it only at the end of work), memory fragmentation is not a typical problem and it is likely that the rate of management of memory allocated from a heap turns out to be higher. The authors do not insist upon this hypothesis but still have no another one.

*Mpi_leave_pinned* being off in the OpenMPI package gives the third curve in Fig. 1.

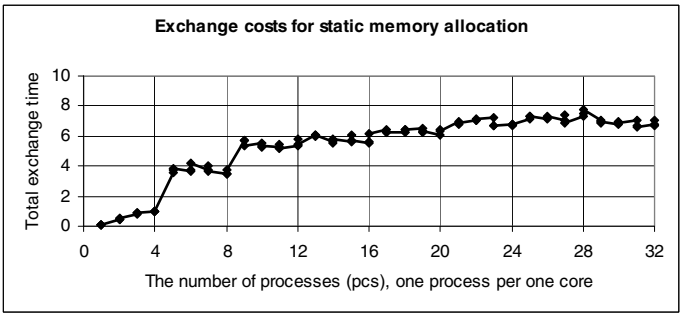



**Fig. 2.** Graph of dependence of exchange time with synchronization on the number of computational processes (one process per core) for SMP-node two-cores two-processors per host cluster architecture

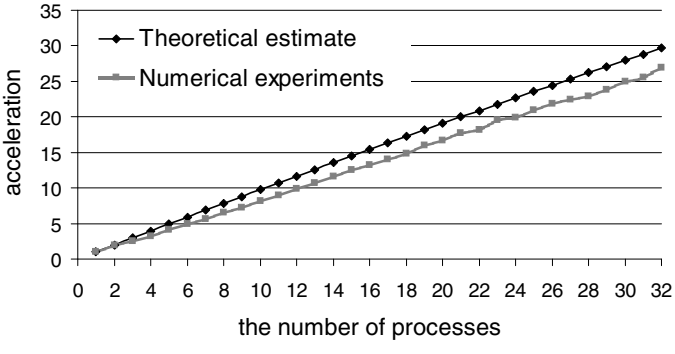



**Fig. 3.** Dependence of acceleration on the number of nodes

Thus, the study shows that dynamic memory allocation without correction of strategy provides the worst results in rate as well as in smoothness of the time expense curve. We notice that the results for both packages coincide with high degree of accuracy. In Fig. 2, graphs of communication time for the problem are shown. Exchange time differs only slightly for all strategies, therefore only one graph is demonstrated. At the point $p = 4$ it has an expected jump due to putting into operation a network of data transfer for exchanges between processors 3 and 4 which are found to be different hosts. Further we observe a less evident but also explicable jump at the point $p = 8$ where the host carrying processes $4 - 7$ begins to exchange with two external neighbors (the third and eight ones) over the same network. Then the number of external neighbors no longer increases, and we see slight smooth increase of exchange time on the graph. The last fact is related to the need to perform exchanges of the total value of discrete uniform norm, and cost of this procedure depends on the number of processes although this dependence is weak.

Fig. 3 illustrates dependence of acceleration of computations on the number of processes for an $801 \times 801$ grid obtained with the cluster of ICM SB RAS after adjustment of strategy in the case of dynamic memory allocation. For comparison, a graph of potential acceleration according to the estimate (5) is presented. Calculations performed with a cluster demonstrate a classical pattern of acceleration which confirms that it increases with efficiency about one (efficiency of calculations for 32 nodes $\approx 0.70$).

**Acknowledgments.** The authors are grateful to Andrey V. Malyshev.

## References

1. Marchuk, G.I., Kagan, B.A.: Dynamics of Ocean Tides. Leningrad, Gidrometizdat (1983)
2. Agoshkov, V.I.: Inverse problems of the mathematical theory of tides: boundary-function problem. Russ. J. Numer. Anal. Math. Modelling 20(1), 1–18 (2005)
3. Kamenshchikov, L.P., Karepova, E.D., Shaidurov, V.V.: Simulation of surface waves in basins by the finite element method. Russian J. Numer. Anal. Math. Modelling 21(4), 305–320 (2006)
4. Karepova, E.D., Shaidurov, V.V.: Parallel implementation of FEM for the initial boundary value problem. Computational Technologies 14(6), 45–57 (2009) (in Russian)
5. Kireev, I.V., Pyataev, S.F.: Pixel technologies of discretization of a water area of the World Ocean. Computational Technologies 14(5), 30–39 (2009) (in Russian)

# A Parallel Implementation of GaussSieve for the Shortest Vector Problem in Lattices

Benjamin Milde and Michael Schneider

Technische Universität Darmstadt, Germany
mischnei@cdc.informatik.tu-darmstadt.de

**Abstract.** The security of lattice based cryptography can be considered to be based on the hardness of the shortest vector problem (SVP) in lattices. Sieving algorithms can be used to solve this problem, at least in small dimensions. The most promising among the sieving algorithms is GaussSieve.

In this paper we present a parallel version of the GaussSieve algorithm that solves the shortest vector problem in lattices. For small number of up to 5 parallel threads, the parallel version scales nearly linearly. For bigger numbers of threads, the efficiency decreases. We implement the parallel GaussSieve on multicore CPUs, whereas the presented ideas can also be implemented on different parallel platforms.

**Keywords:** Shortest Vector Problem (SVP), GaussSieve, Parallelization, Multicore CPU.

## 1 Introduction

A lattice is a discrete additive subgroup of $\mathbb{R}^n$. Elements of a lattice are called vectors. A lattice can be represented by its basis, a set of linearly independent vectors $\boldsymbol{b}_i \in \mathbb{R}^n$. The *shortest vector problem* (SVP) is stated the following: given a basis of a lattice, output a shortest non-zero element of the lattice. In cryptography, hard problems in lattices like the shortest vector problem are used as basis of digital signatures, encryption schemes, hash functions, and many more cryptographic primitives. Therefore, examining the shortest vector problem implies the ability to assess the hardness of lattice based cryptosystems.

Basically there are three approaches to solve SVP. First, there are enumeration algorithms [FP83, SE94, GNR10]. They perform exhaustive search among all coefficient vectors in a specified search region, using branch-and-bound strategy in a search tree. Second, there are algorithms applying the computation of Voronoi cells [MV10a]. Third, there are probabilistic sieve algorithms, that output a shortest vector with high probability [AKS01, NV08]. The GaussSieve algorithm that is dealt with in this paper belongs to this third category. It was presented in [MV10b] and can be seen as the fastest sieving algorithm as of today. Therefore, we have chosen GaussSieve as the basis of our parallel sieve algorithm.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 452–458, 2011.

Parallel versions of enumeration are known for graphics cards [HSB+10] and FPGAs [DHPS10]. To our knowledge, there is no previous work concerning the parallelization of sieve algorithms for SVP. Our multicore implementation is the first attempt of parallelizing GaussSieve. It shows the strengths and the limits of parallelization for this kind of algorithms.

## 2 The GaussSieve Algorithm

The GaussSieve algorithm was presented in [MV10b]. It stores a list $L$ of lattice vectors, that are *Gauss-reduced* against each other. The list grows by adding randomly sampled and reduced vectors to the list $L$. After a sufficient number of vectors has been added, one of the list vectors is a shortest lattice vector with high probability. A stack $S$ is used to keep the list $L$ small. As soon as a vector from the list can be reduced, it is pushed to the stack $S$. Stack vectors are chosen in the next round instead of sampling new vectors.

If started with a goal norm as input parameter, GaussSieve terminates as soon as it has found a vector of size below the desired bound. If a new vector is linearly dependent from list vectors, it will be reduced to the zero vector. This is called a collision. After a pre-defined number of collisions the algorithm terminates and outputs the shortest vector found so far.

Algorithm 1 shows a pseudo-code description of GaussSieve. The functions GaussReduce and Reduce are presented as Algorithms 2 and 3. For further information we refer to the original paper [MV10b].

**Algorithm 1.** GaussSieve

**Input**: Lattice basis $\mathbf{B}$, BKZ blocksize $\beta$, targetNorm, maxCollision
**Output**: $\boldsymbol{v}_{best}$: shortest vector found

List L $\leftarrow BKZ_\beta(B)$
$K \leftarrow 0$
$\boldsymbol{v}_{best} \leftarrow \boldsymbol{l}_{min}$ where $\langle \boldsymbol{l}_{min}, \boldsymbol{l}_{min} \rangle = \min_{\boldsymbol{l}_i \in L} \langle \boldsymbol{l}_i, \boldsymbol{l}_i \rangle$
$currentBestNorm \leftarrow \langle \boldsymbol{v}_{best}, \boldsymbol{v}_{best} \rangle$
**while** $currentBestNorm > targetNorm \wedge K < maxCollision$ **do**
    **if** *S.empty()* **then**
        $\boldsymbol{v}_{new} \leftarrow$ sampleRandomLatticeVector(B)
    **else**
        $\boldsymbol{v}_{new} \leftarrow$ S.pop()
    **end**
    GaussReduce($\boldsymbol{v}_{new}$, L, S)
    squarenorm $\leftarrow \langle \boldsymbol{v}_{new}, \boldsymbol{v}_{new} \rangle$
    **if** $squarenorm = 0$ **then**
        K $\leftarrow$ K + 1 //this is a collision
    **else**
        **if** $squarenorm < currentBestNorm$ **then**
            currentBestNorm $\leftarrow$ squareNorm
            $\boldsymbol{v}_{best} \leftarrow \boldsymbol{v}_{new}$
        **end**
        L.insert($\boldsymbol{v}_{new}$)
    **end**
**end**
**return** $\boldsymbol{v}_{best}$

**Algorithm 2.** GaussReduce

```
   Input: p, L, S
 1 reduceFurther ← true
 2 while reduceFurther do
 3 |   reduceFurther ← false
 4 |   for v ∈ L where ⟨v, v⟩ ≤ ⟨p, p⟩ do
 5 |   |   if Reduce(p, v) then
 6 |   |   |   reduceFurther ← true
 7 |   |   end
 8 |   end
 9 end
10 for v ∈ L where ⟨v, v⟩ > ⟨p, p⟩ do
11 |   if Reduce(v, p) then
12 |   |   S.push(v)
13 |   |   L.erase(v)
14 |   end
15 end
```

**Algorithm 3.** Reduce

```
   Input: p, v
 1 if abs(2 · ⟨p, v⟩) > ⟨v, v⟩ then
 2 |   mul ← round (⟨p,v⟩ / ⟨v,v⟩)
 3 |   p -= mul · v
 4 |   return true
 5 end
 6 return false
```

It is common to use the BKZ algorithm (controlled by a blocksize-parameter $\beta$) [SE94] to pre-reduce the basis. Our pseudo-code contains some improvements over the original algorithm. The list $L$ is sorted according to the square-norms of the vectors. This speeds up computation in the GaussReduce part, since it is then simpler to iterate over a certain sub-group of vectors of specific norm. Instead of subtracting each vector multiple times, the multiplier in the *Reduce*-function is used. Both improvements were not explained in the GaussSieve paper, but were already contained in the original implementation of GaussSieve [Vou10], which is publicly available.

## 3 Parallel GaussSieve

The idea of the parallel version of GaussSieve is to extend the single GaussSieve algorithm into a network parallel variant. Several independent GaussSieve instances are connected together in a ring fashion, so that a vector that passes the list of one instance is handed over to the next instance. A queue is added to the system that acts like a buffer, and the vectors are handed over to the buffer of the next instance. Figure 1 illustrates this approach.

If a vector passed all instances, it is added to the list of the current instance. All vectors are marked by the number of the instance that initiated the vector, so that each instance can recognize a vector that already passed all instances. Such a vector is nearly pair-wise reduced to all lists, small changes can occur since lists that are already passed could change while the vector is still in the system.

Each instance uses vectors from the queue with priority, so that vectors that are still in the system get processed quickly. If the queue is empty, the stack is used, and like in the original algorithm, if the stack is empty, new vectors are sampled. Also like in the single GaussSieve, lists grow and shrink. However, this parallel variant regulates the list sizes automatically, so that they are approximately the same size among all instances: Vectors are added to the list, if an

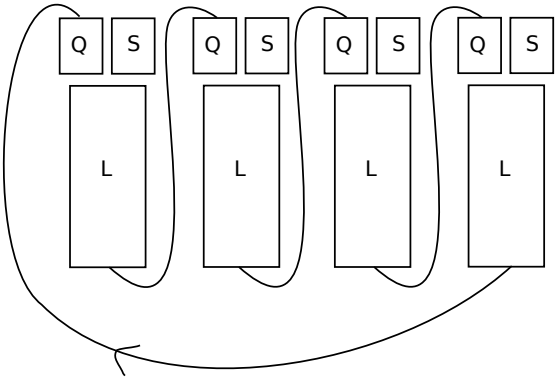



**Fig. 1.** This figure illustrates the idea of a network parallel variant of GaussSieve. Each instance of GaussSieve processes the vectors of its queue $Q$ and hands processed vectors over to the next instance. The stacks $S$ are filled with list vectors that were reduced and need to pass all lists $L$ once more.

instance's queue is empty and it can spawn new vectors into the system from the stack or by sampling. If a list is too small and vectors get processed quicker than in other instances, more new vectors will be spawned and sooner or later the list grows. If the list is too big compared to the other instances, no new vectors will be spawned because the queue is never empty and the list size is reduced by vectors that are removed from the list and put on the stack, until the queue gets empty again.

To prevent hotspot situations and to let the system respond quicker to imbalances in list sizes, the queues are restricted to a maximum size. If the maximum size is exceeded and the vector does not belong to this particular instance, it can jump over the full instance and skip it. This way a jam of vectors in a particular instance is prevented that would stop the whole system from working. In a network parallel setup, an instance that does not respond anymore could be also skipped this way, so that a failure in a particular instance is not critical to the overall system. But this means also that not all vectors are pair-wise reduced and that the algorithm behaves differently from the single GaussSieve. This can be accommodated by processing more vectors in total (also increasing run time). The more instances participate in this network parallel version, the more overall additional run time must be spend, decreasing the efficiency compared to the single non-parallel GaussSieve.

## 4 Experimental Results

The random lattices from the SVP challenge [GS10] have been used to make accurate measurements about timings and efficiency of our parallel GaussSieve variant. They are constructed in the sense of Goldstein and Mayer [GM03]. These lattices show a nice random behaviour and are kind of standard lattices

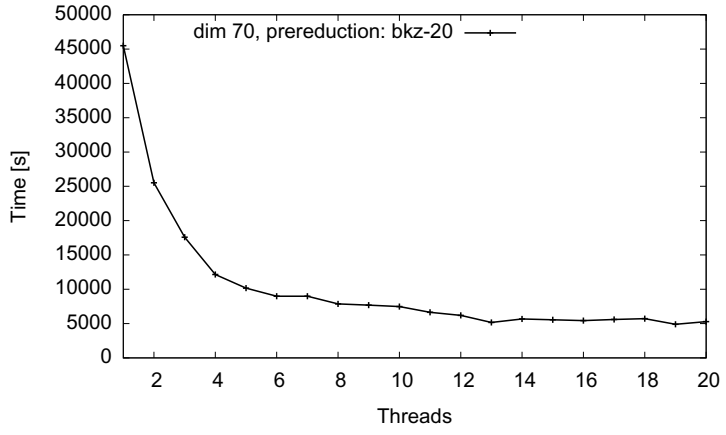



**Fig. 2.** Timings for GaussSieve of a lattice with dimension 70 and BKZ pre-reduction of $\beta = 20$

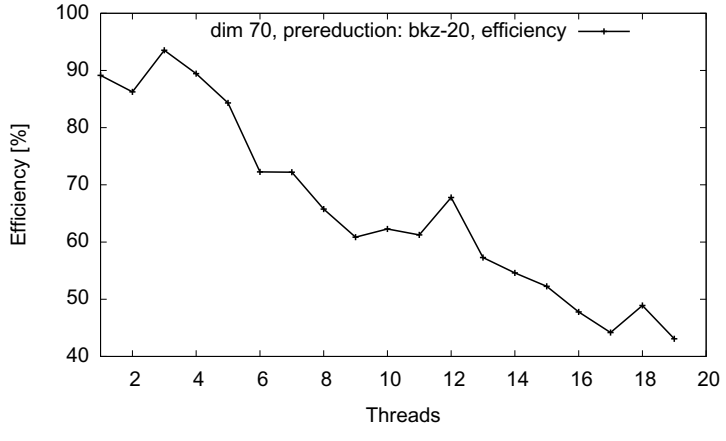



**Fig. 3.** Efficiency for the parallel version compared to a single GaussSieve, for a 70-dimensional lattice and a BKZ pre-reduction of $\beta = 20$. The efficiency decreases linearly, this is due to not complete pair-wise reduced lists. However results are still very promising in a 1-10 thread parallel setting.

for testing SVP algorithms. We present timings for the 70-dimensional lattice, which shows a typical behaviour. Figure 2 shows run times for 1-20 parallel threads measured on a machine equipped with four 2,6 GHz Istanbul Opteron Workstation CPUs. By adding more parallel computing power, overall time can be decreased, however efficiency decreases linearly as the amount of available threads/cores increases, see Figure 3. The reason for this behaviour is that more vectors need to be processed as the parallel computing power increases, which can be seen in Figure 4.

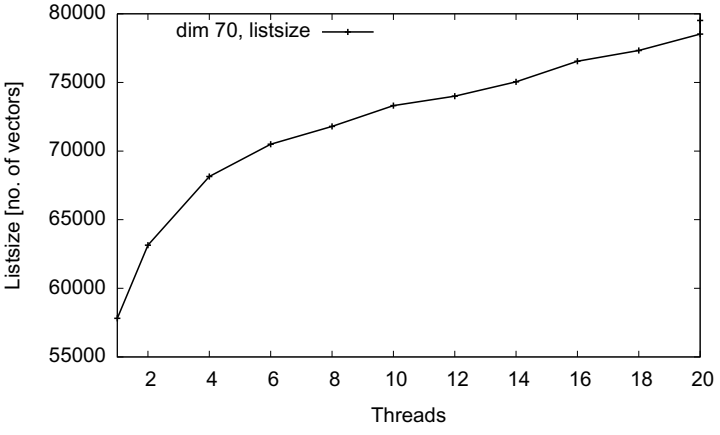



**Fig. 4.** Since not all vectors are pair-wise reduced, more vectors are needed in the lists to find the shortest vector. The graph shows the total number of all stored vectors at the time the shortest vector was found.

## 5 Conclusion and Further Work

From its nature, ListSieve (also presented in [MV10b]) is much more suitable for parallelization than GaussSieve. In ListSieve, the list of vectors is never touched, therefore different threads can easily share one list. It is to be seen if the advantage of parallelization can compensate the disadvantages that ListSieve has compared to GaussSieve. It is possible that ParallelListSieve outperforms ParallelGaussSieve using big numbers of threads.

To our knowledge, the improvement of [PS09] has not been applied for any sieving implementation. This might also improve the efficiency of the algorithm.

**Acknowledgements.** Michael Schneider is supported by project BU 630/23-1 of the German Research Foundation (DFG). This work was supported by CASED (www.cased.de). We thank Chen-Mou Cheng and Bo-Yin Yang for initial discussions on the topic.

## References

[AKS01] Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: STOC 2001, pp. 601–610. ACM, New York (2001)

[DHPS10] Detrey, J., Hanrot, G., Pujol, X., Stehlé, D.: Accelerating Lattice Reduction with FPGAs. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 124–143. Springer, Heidelberg (2010)

[FP83] Fincke, U., Pohst, M.: A procedure for determining algebraic integers of given norm. In: van Hulzen, J.A. (ed.) ISSAC 1983 and EUROCAL 1983. LNCS, vol. 162, pp. 194–202. Springer, Heidelberg (1983)

[GM03] Goldstein, D., Mayer, A.: On the equidistribution of Hecke points. Forum Mathematicum 15(2), 165–189 (2003)

[GNR10] Gama, N., Nguyen, P.Q., Regev, O.: Lattice Enumeration Using Extreme Pruning. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 257–278. Springer, Heidelberg (2010)

[GS10] Gama, N., Schneider, M.: SVP Challenge (2010), http://www.latticechallenge.org/svp-challenge

[HSB+10] Hermans, J., Schneider, M., Buchmann, J., Vercauteren, F., Preneel, B.: Parallel Shortest Lattice Vector Enumeration on Graphics Cards. In: Bernstein, D.J., Lange, T. (eds.) AFRICACRYPT 2010. LNCS, vol. 6055, pp. 52–68. Springer, Heidelberg (2010)

[MV10a] Micciancio, D., Voulgaris, P.: A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. In: STOC. ACM, New York (2010)

[MV10b] Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: SODA, pp. 1468–1480. ACM/SIAM (2010)

[NV08] Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. J. of Mathematical Cryptology 2(2) (2008)

[PS09] Pujol, X., Stehlé, D.: Solving the shortest lattice vector problem in time $2^{2.465n}$. Cryptology ePrint Archive, Report 2009/605 (2009), http://eprint.iacr.org/

[SE94] Schnorr, C.-P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. Mathematical Programming 66, 181–199 (1994)

[Vou10] Voulgaris, P.: Gauss Sieve alpha V. 0.1, Panagiotis Voulgaris' homepage at the University of California, San Diego (2010), http://cseweb.ucsd.edu/~pvoulgar/impl.html

# Graphical Algorithm for the Knapsack Problems

Alexander Lazarev, Anton Salnikov, and Anton Baranov

Institute of Control Sciences of the Russian Academy of Sciences,
Profsoyuznaya st. 65, 117997 Moscow, Russia,
Lomonosov Moscow State University,
State University – Higher School of Economics,
Moscow Institute of Physics and Technology – State University
jobmath@mail.ru,salnikov@ipu.ru,
av.baranov@physics.msu.ru

**Abstract.** We consider a modification of dynamic programming algorithm ($DPA$), which is called as graphical algorithm ($GA$). For the knapsack problem ($KP$) it is shown that the time complexity of $GA$ is less than the time complexity of $DPA$. Moreover, the running time of $GA$ is often essentially reduced. $GA$ can also solve big scale instances and instances, where the parameters are not only positive integer. The paper outlines different methods of parallelizing $GA$ taking into account its main features and advantages to various parallel architectures, in particular by using $OpenCL$ and $MPI$ framework. Experiments show that "hard" instances of $KP$ for $GA$ have correlation $p_j \simeq kw_j$ for all $j$, where $p_j$ and $w_j$ are utility and capacity of item $j = 1, 2, \ldots, n$.

**Keywords:** Graphical Algorithm, Knapsack Problem, OpenCL, MPI, Parallel Algorithm.

## 1 Introduction

Dynamic programming is a general optimization technique developed by Bellman [1]. It can be considered as a recursive optimization procedure which interprets the optimization problem as a multi-step solution process. Bellman's optimality principle can be briefly formulated as follows: Starting from any current step, an optimal policy for the subsequent steps is independent of the policy adopted in the previous steps. In the case of a combinatorial problem, at some step $j$, $j = 2, \ldots, n$, sets of a particular size $j$ are considered. To determine the optimal criterion value for a particular subset of size $j$, one has to know the optimal values for all necessary subsets of size $j - 1$. If the problem includes $n$ elements, the number of subsets to be considered is equal to $O(2^n)$. Therefore, dynamic programming usually results in an exponential complexity. However, if the problem considered is $NP$-hard in the ordinary sense, it is possible to derive pseudo-polynomial algorithms [2,3,4].

In this paper, we give the basic idea of a graphical modification of dynamic programming algorithm ($DPA$), which is called Graphical Algorithm ($GA$). This approach often reduces the number of it states to be considered in each step of

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 459–466, 2011.

a $DPA$. Moreover, in contrast to classical $DPA$, it can also treat problems with non-integer data without necessary transformations of the corresponding problem. In addition, for some problems, $GA$ essentially reduces the time complexity.

For the knapsack problem DPA with the same idea like in $GA$ are known (e.g. see [7]). In such $DPA$ not all states $t \in [0, C]$ are considered, but only states, where a value of objective function is changed. Thus, the time complexity of such $DPA$ is bounded by $O(nF_{opt})$, where $F_{opt}$ is the optimal value of objective function. However, these algorithms can be useful only for problems, where $F_{opt} < C$, otherwise we can use the classical $DPA$. We generalize the idea of such algorithms for the objective function, for which $F_{opt} \gg C$.

This paper is organized as follows. In Section 2, we give the basic idea of the $GA$. In the next section we describe graphical algorithm for the binary knapsack problem. Section 4 describes parallel implementation of $GA$ using $OpenCL$ and $MPI$ framework. Last section represents the results of experiments to search for and analyse of "hard" examples.

## 2 Basic Idea of the Graphical Algorithm

Usually in $DPA$, we have to compute the value $f_j(t)$ of a particular function for each possible state $t$ at each stage $j$ of a decision process, where $t \in [0, C]$ and $t, C \in Z^+$. If this is done for any stage $j = 1, 2, \ldots, n$, where $n$ is a size of the problem, the time complexity of such a $DPA$ is typically $O(nC)$. However, often it is not necessary to store the result for any integer state since in the interval $[t_l, t_{l+1})$, we have a functional equation $f_j(t) = \varphi(t)$ (e.g. $f_j(t) = k_j \cdot t + b_j$, i.e., $f_j(t)$ a continuous linear function when allowing also real values $t$).

Assume that we have the following functional equations in a $DPA$, which correspond to Bellman's recursive equations:

$$f_j(t) = \min_{j=1,2,\ldots,n} \begin{cases} \Phi^1(t) = \alpha_j(t) + f_{j-1}(t - w_j) \\ \Phi^2(t) = \beta_j(t) + f_{j-1}(t - b_j) \end{cases} \tag{1}$$

with the initial conditions

$$\begin{aligned} f_0(t) &= 0, \qquad \text{for } t \geq 0, \\ f_0(t) &= +\infty, \qquad \text{for } t < 0. \end{aligned} \tag{2}$$

In (1), function $\Phi^1(t)$ characterizes a setting $x_j = 1$ while $\Phi^2(t)$ characterizes a setting $x_j = 0$ representing a yes/no decision, e.g. for an item, a job [2],[6]. In step $j$, $j = 1, 2, \ldots, n$, we compute and store the data given in Table 1.

Here $X(y), y = 0, 1, \ldots, C$, is a vector which describes an optimal partial solution and which consists of $j$ elements (values) $x_1, x_2, \ldots, x_j \in \{0, 1\}$.

**Table 1.** Computations in $DPA$

| $t$ | 0 | 1 | 2 | ... | $y$ | ... | $C$ |
|---|---|---|---|---|---|---|---|
| $f_j(t)$ | $value_0$ | $value_1$ | $value_2$ | ... | $value_y$ | ... | $value_C$ |
| optimal partial solution $X(t)$ | $X(0)$ | $X(1)$ | $X(2)$ | ... | $X(y)$ | ... | $X(C)$ |

**Table 2.** Computations in $GA$

| $t$ | $[t_0, t_1)$ | $[t_1, t_2)$ | $\ldots$ | $[t_l, t_{l+1})$ | $\ldots$ | $[t_{m_j-1}, t_{m_j}]$ |
|---|---|---|---|---|---|---|
| $f_j(t)$ | $\varphi_1(t)$ | $\varphi_2(t)$ | $\ldots$ | $\varphi_{l+1}(t)$ | $\ldots$ | $\varphi_{m_j}(t)$ |
| optimal partial solution $X(t)$ | $X(t_0)$ | $X(t_1)$ | $\ldots$ | $X(t_l)$ | $\ldots$ | $X(t_{m_j-1})$ |

However, this data can also be stored in a condensed tabular form as given in Table 2.

Here, we have $0 = t_0 < t_1 < t_2 < \ldots < t_{m_j} = C$.

To compute function $f_{j+1}(t)$, we compare two temporary functions $\Phi^1(t)$ and $\Phi^2(t)$.

The function $\Phi^1(t)$ is a combination of the terms $\alpha_{j+1}(t)$ and $f_j(t - w_{j+1})$. Function $f_j(t - w_{j+1})$ has the same structure as in Table 2, but all intervals $[t_l, t_{l+1})$ have been replaced by $[t_l - w_{j+1}, t_{l+1} - w_{j+1})$, i.e., we shift the graph of function $f_j(t)$ to the right by the value $w_{j+1}$. If we can present function $\alpha_{j+1}(t)$ in the same form as in Table 2 with $\mu_1$ columns, we store function $\Phi^1(t)$ in the form of Table 2 with $m_j + \mu_1$ columns. In an analogous way, we store function $\Phi^2(t)$ in the form of Table 2 with $m_j + \mu_2$ columns.

Then we construct function

$$f_{j+1}(t) = \min\{\Phi^1(t), \Phi^2(t)\}.$$

For example, let the columns of Table $\Phi^1(t)$ contain the intervals

$$[t_0^1, t_1^1), [t_1^1, t_2^1), \ldots, [t_{(m_j+\mu_1)-1}^1, t_{(m_j+\mu_1)}^1]$$

and the columns of Table $\Phi^2(t)$ contain the intervals

$$[t_0^2, t_1^2), [t_1^2, t_2^2), \ldots, [t_{(m_j+\mu_2)-1}^2, t_{(m_j+\mu_2)}^2].$$

To construct function $f_{j+1}(t)$, we compare the two functions $\Phi^1(t)$ and $\Phi^2(t)$ on each interval, which is formed by means of the points

$$\{\, t_0^1, t_1^1, t_2^1, \ldots, t_{(m_j+\mu_1)-1}^1, t_{(m_j+\mu_1)}^1, \\ t_0^2, t_1^2, t_2^2, \ldots, t_{(m_j+\mu_2)-1}^2, t_{(m_j+\mu_2)}^2 \},$$

and we determine the intersection points $t_1^3, t_2^3, \ldots, t_{\mu_3}^3$. Thus, in the table of function $f_{j+1}(t)$, we have at most $2m_j + \mu_1 + \mu_2 + \mu_3 \leq C$ intervals.

In fact, in each step $j = 1, 2, \ldots, n$, we do not consider all points $t \in [0, C]$, $t$, $C \in Z^+$, but only points from the interval in which the optimal partial solution changes or where the resulting functional equation of the objective function changes. For some objective functions, the number of such points $M$ is small and the new algorithm based on this graphical approach has a time complexity of $O(n \min\{C, M\})$ instead of $O(nC)$ for the original $DPA$.

Moreover, such an approach has some other advantages.

1. The $GA$ can solve instances, where $p_j$, $w_j$, $j = 1, 2, \ldots, n$, or/and $C$ are not integer.

2. The running time of the $GA$ for two instances with the parameters $\{p_j, w_j, C\}$ and $\{p_j \cdot 10^\alpha \pm 1,\ w_j \cdot 10^\alpha \pm 1, C \cdot 10^\alpha \pm 1\}$ is the same while the running time of the $DPA$ will be $10^\alpha$ times larger in the second case. Thus, using the $GA$, one can usually solve considerably larger instances.
3. Properties of an optimal solution are taken into account. For KP, an item with the smallest value $\frac{p_j}{w_j}$ may not influence the running time.
4. As we will show below, for several problems, $GA$ has even a polynomial time complexity or we can at least essentially reduce the complexity of the standard $DPA$.

Thus, the use of $GA$ can reduce both the time complexity and the running time for $KP$. The application of $GA$ to the partition problem is described in detail in [5], where also computational results are presented.

## 3 Graphical Algorithm for the Knapsack Problem

In this section, we describe the application of this approach to the one-dimensional knapsack problem [5].

**One-dimensional knapsack problem ($KP$):** One wishes to fill a knapsack of capacity $C$ with items having the largest possible total utility. If any item can be put at most once into the knapsack, we get the binary or $0-1$ knapsack problem. This problem can be written as the following integer linear programming problem:

$$\begin{cases} f(x) = \sum\limits_{j=1}^{n} p_j x_j \to \max \\ \sum\limits_{j=1}^{n} w_j x_j \le C, \\ x_j \in \{0,1\},\ j = 1,2,\ldots,n. \end{cases} \tag{3}$$

Here, $p_j$ gives the utility and $w_j$ the required capacity of item $j$, $j = 1,2,\ldots,n$. The variable $x_j$ characterizes whether item $j$ is put into the knapsack or not.

The $DPA$ based on Bellman's optimality principle is one of the standard algorithms for the $KP$. It is assumed that all parameters are positive integer: $C, p_j, w_j \in Z^+, j = 1,2,\ldots,n$.

For $KP$, Bellman's recursive equations are as follows:

$$f_j(t) = \max_{j=1,2,\ldots,n} \begin{cases} \Phi^1(t) = p_j + f_{j-1}(t - w_j) \\ \Phi^2(t) = f_{j-1}(t), \end{cases} \tag{4}$$

where

$$f_0(t) = 0, \quad t \ge 0,$$
$$f_0(t) = +\infty,\ t < 0.$$

$\Phi^1(t)$ represents the setting $x_j = 1$ (i.e., item $j$ is put into the knapsack) while $\Phi^2(t)$ represents the setting $x_j = 0$ (i.e., item $j$ is not put into the knapsack). In each step $j$, $j = 1,2,\ldots,n$, the function values $f_j(t)$ are calculated for each integer point (i.e., "state") $0 \le t \le C$. For each point $t$, a corresponding best (partial) solution $X(t) = (x_1(t), x_2(t), \ldots, x_j(t))$ is stored.

## 4 Parallel Implementation

This section describes the parallel implementation of $GA$ using $OpenCL$ framework and $MPI$.

### 4.1 Column Parallelization

The classical Bellman's recurrence (4) can be implemented by creating an array with $C + 1$ rows and $n$ columns to store all the values in each step of the program that sequentially adds items into the problem by filling up the table column by column, where $C$ – is capacity of the knapsack. To compute the values for any column the values from the previous column are only needed. Such economical consumption of memory allows us easily implement this algorithm using a variety of parallel programming models including shared memory and distributed memory $API$s.

The specificity of $GA$ allows us to renounce the use of the columns of the same length $C$ in each step of the host program, and gradually increase the number of parallel processors while we add items to the problem. For the $GA$ "length" of the column in each step depends on the distance between the boundary break points. Furthermore, taking into account this feature we could essentially reduce the running time of the program by initial sorting the items in non-decreasing order of values $w_j$.

Fig. 1, 2 show plots of the number of break-points for $GA$ and $DPA$ in the two cases: when $C$ is chosen at a rate of 10% and 90% of the $C_{\max} = \sum_{i=1}^{n} w_i$.

### 4.2 Interval Parallelization

This parallelization is not easy to implement but it allows us to take main advantage of the $GA$.

When we look closer to the $GA$ we note that to create a table of intervals in step $j$, $j = 2, \ldots, n$, of the sequential program all we need are values from the

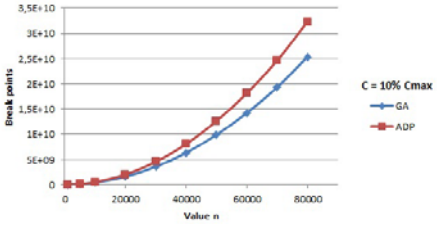



**Fig. 1.** The dependence of the number of break points on the number of items. $C = 10\%C_{\max}$.

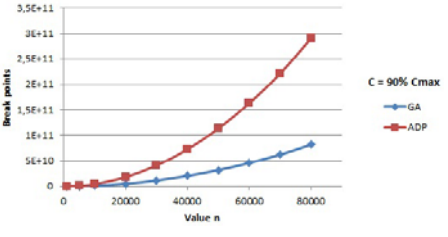



**Fig. 2.** The dependence of the number of break points on the number of items. $C = 90\%C_{\max}$.

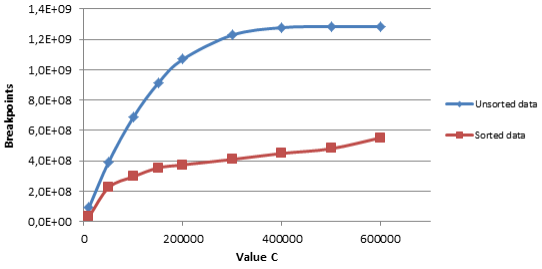



**Fig. 3.** The dependence of the number of break-points on the value of $C$ for sorted and unsorted initial data

previous table in step $j-1$. Therefore in each step we can partition the table of intervals from the previous step into columns, and make each parallel processor responsible for only one column i.e. one interval between two break points. Then each processor has to update information about break points and corresponding intervals for the table in the current step.

This is quite a tricky procedure because we could obtain one, two or none current intervals depending on situation for each previous interval. Furthermore, every parallel processor should have access to the table of intervals to calculate the new values. The last obstacle is easily bypassed on shared memory models but the first one requires additional memory and computational resources to store and extract the temporal information given by every parallel processor for the current table of intervals in each step of the sequential program. The practical realization of this procedure depends on the chosen data structure.

When parallelizing intervals we also should note that in each step of the sequential program the number of intervals may be doubled, so the partition of the table of intervals in a coarse-grained manner without regular reassigning columns to the new parallel processors could lead to the explosive load on one or several parallel processors and full stop of the program.

While solving large instances it is helpful to control the load of the grid by initial sorting the items in non-increasing order of values $\frac{p_j}{w_j}$. Fig. 3 shows the plots for examples with dimension $n = 10000$.

## 5 Experiments

The main objective of experiments was to search and analyse "hard" examples for which graphical algorithm would show the maximal time complexity. The basic unit of $GA$ is the break point. More break points we have are for more complex problem we encounter. Thus, the main objective of our programming activities was to create the procedure for finding instances with so many break points as possible.

It is obvious that the theoretical maximum number of break points is $2^{n+1}-1$. To obtain this result in real example on positive integers the value of $C$ should

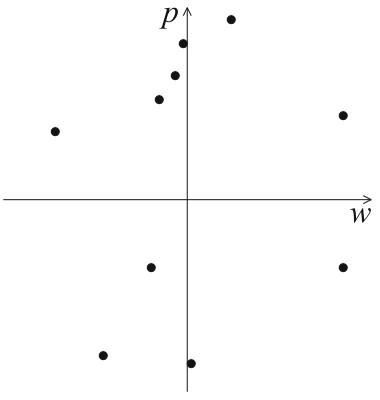



**Fig. 4.** Initial instance, n=10, B=37

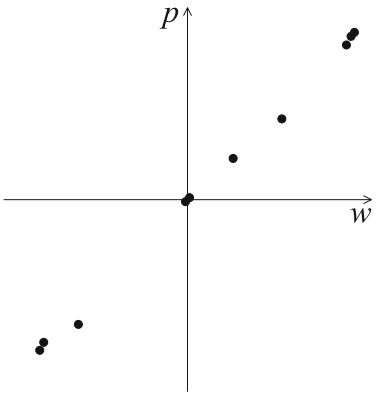



**Fig. 5.** Hard instance, n=10, B=2047

be more than maximum number of break points or we should simply exclude $C$ and associated restrictions from the problem. Additionally, to study the $GA$ deeply we should totally switch into integers by allowing negative values for $w_j$ and $p_j$.

As mentioned before the maximum number of break points that we can get is $2^{n+1} - 1$. This is the most complicated example that is practically impossible to get randomly as it turned out during experiment which instead of it was giving us less than $n^2$ break points for the most of the times. After a series of experiments we came up to some heuristic procedure that could increase the number of break points through the gradual change of the parameters of the initial randomly generated instances.

Our method, despite its apparent simplicity, has shown to be highly effective in the rapid searching for hard instances for which the number of break points is approaching to the maximum value $2^{n+1} - 1$. Fig. 4 and Fig. 5 show first and last stages of this process correspondingly. The experiment was carried out for $n = 10$, $w_j$ and $p_j$ are integers drawn from the normal distribution within interval [-1024, 1024], and $B$ is number of break points.

It is easy to look that all points in Fig. 5 lie very close to the line passing through the center of coordinates $(0, 0)$. This property became apparent in all our experiments, which gives us the right to predicate that all hard instances of $KP$ for $GA$ should satisfy the following correlation $p_j \simeq kw_j$, $j = 1, \ldots, n$.

In summary the most complex instance of $KP$ for $GA$ can be written as follows:

$$\begin{cases} \sum\limits_{j=1}^{n} k \cdot w_j x_j \to \max \\ \sum\limits_{j=1}^{n} w_j x_j \le C, \\ x_j \in \{0, 1\},\ j = 1, 2, \ldots, n. \end{cases} \tag{5}$$

## 6 Concluding Remarks

The graphical approach can be applied to problems where a pseudo-polynomial algorithm exists and Boolean variables are used in the sense that yes/no decisions have to made (e.g. in the problem under consideration, for $KP$, an item can be put into the knapsack or not), for example for partition and scheduling problems. However, for the knapsack problem, the graphical algorithm mostly reduces substantially the number of points to be considered but the time complexity of the algorithm remains pseudo-polynomial.

**Acknowledgements.** Partially supported by programs of Russian Academy of Sciences 15 and 29. We would like to thank Prof. Dr. Frank Werner and Dr. Evgeny Gafarov for many discussions and helpful suggestions.

## References

1. Bellman, R.: Dynamic Programming. Princeton Univ. Press, Princeton (1957)
2. Gafarov, E.R., Lazarev, A.A., Werner, F.: Algorithms for Some Maximization Scheduling Problems on a Single Machine. Automation and Remote Control 71(10), 2070–2084 (2010)
3. Keller, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Hidelberg (2010)
4. Lawler, E.L., Moore, J.M.: A Functional Equation and its Application to Resource Allocation and Sequencing Problems. Management Science 16(1), 77–84 (1969)
5. Lazarev, A.A., Werner, F.: A Graphical Realization of the Dynamic Programming Method for Solving $NP$-Hard Combinatorial Problems. Computers and Mathematics with Applications 58(4), 619–631 (2009)
6. Lazarev, A.A., Werner, F.: Algorithms for Special Cases of the Single Machine Total Tardiness Problem and an Application to the Even-Odd Partition Problem. Mathematical and Computer Modelling 49(9-10), 2061–2072 (2009)
7. Papadimitrou, C., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, INC., New York (1998)

# SSCCIP – A Framework for Building Distributed High-Performance Image Processing Technologies*

Evgeny V. Rusin

Institute of Computational Mathematics and Mathematical Geophysics SB RAS
prospect Akademika Lavrentjeva, 6, Novosibirsk, 630090, Russia
rev@ooi.sscc.ru

**Abstract.** The paper describes the experimental framework for distributed image processing with the use of multicomputer providing fast development of high-performance remote sensing data processing technologies. Basic principles of system building, some architectural solutions, and sample implementation of concrete processing technologies are given.

**Keywords:** remote sensing, image processing, parallel computations, distributed processing, computational technologies.

## 1 Introduction

Remote sensing data processing tasks are characterized by the huge amount of data to process ($10^8$ multispectral pixels is a typical case) and high labor-intensiveness of processing algorithms (can easily exceed $10^8$ operations per pixel). The need for remote sensing data real-time analysis and interpretation (for example in flood or forest fire monitoring) causes the necessity of using high-performance computational tools. The most common type of such tools is multicomputers, MIMD systems with distributed memory. Involvement of multicomputers to the remote sensing data processing presently leads to the problem of supercomputer "hostility" to user. Typically, multicomputer works under UNIX-like operating system and acts as a computational server interacting with its clients via secured SSH channel; but widespread SSH access software (like PuTTy or SSH Secure Shell) provides user only with text console service. Thus, using multicomputers turns out to be available for those users only who are experienced in UNIX systems and very inconvenient to average users of MS Windows family systems (in some countries, for example in Russia, MS Windows is installed on overwhelming majority of all PCs). And even after this, data processing on multicomputer is carried out in several steps: uploading data to multicomputer with SFTP client (SFTP – Secure FTP – secured protocol for data transfer over SSH), launching processing command on multicomputer with SSH terminal, and downloading results from multicomputer with same SFTP client.

Thus, for mass involvement of high-performance computers to the solution of time-consuming image processing problems, it is necessary not only to create

* Supported by Russian Foundation for Basic Research (project No. 10-07-00131a).

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 467–472, 2011.

software implementing wide range of image processing methods and algorithms on multiprocessor systems (survey of researches of this kind can be found in [1]), but also to create technologies simplifying complicated for average user access to remote multiprocessor UNIX computer.

In the paper we consider an attempt to solve this problem, the experimental SSCCIP (Siberian Scientific Computing Center – Image Processing) system created in the Institute of Computational Mathematics and Mathematical Geophysics SB RAS.

## 2 Requirements

We formulated the following requirements for SSCCIP software:

- The system shall be operator workstation software under MS Windows providing the service of high-performance remote sensing data processing on remote multicomputer.
- The system shall provide the visualization of input and output data.
- The system shall provide the possibility to form processing task interactively.
- The system shall provide transparent for operator execution of processing task on remote multicomputer.
- The system shall be customizable to concrete computational environment (multicomputer, computational library, task queue management, etc.).
- The system shall provide maximal easiness of extensibility with new algorithms.

The latter requirement is especially important because the ease of extension and customization of such systems for the needs of concrete users are the necessary conditions of their further development.

## 3 Composition

SSCCIP computing system consists of three main parts:

- Client component operating on client computer under operator's control.
- Server component operating on remote multicomputer and carrying out calculations proper.
- Communication component securely connecting the client and the server ones.

### 3.1 Client-Server Interaction

The vast majority of multicomputer servers accept only secured SSH client connections. What provides such a connection in SSCCIP system is:

- On the server side, system SSH daemon supporting applied SSH-based protocols SFTP (for file transfer) and rexec (for remote execution of commands/programs on the server).

- On the client side, CryptLib library by Peter Gutmann from Auckland University, New Zeeland [2]. On the CryptLib basis, we implemented SFTP and rexec applied protocols.

On the higher level, the client component interacts with the server by the following scheme:

1. Client component uploads source files and the special file containing task description including specification of the names of source and result files, processing algorithms, and their parameters (hereafter "task file") to the server.
2. Client component launches (enqueues parallel task) server component on the remote multicomputer and waits for calculations completion (polls task queue with some time interval).
3. Server component parses task file, performs the specified operation on the specified data, and stores results in the specified files.
4. After finishing calculations, client component downloads processing results from the server to client computer and visualizes them.

### 3.2 Client Component

Client component is a MS Windows application SSCCIP_Client implemented in C++. Its source code is organized as a framework providing the ease of introducing new processing operations: code performing the actions which are common for all processing operations (uploading source data and task files to server, launching and tracking task, downloading results from server) forms `executeParallelTask()` generic subprogram; to code a concrete processing operation, one need to implement only the actions which are specific for the operation: interactive configuration (source and result files selection, processing algorithm parameterization) and visualization of the results. For this aim framework declares `IParallelTask` interface with two abstract methods, `Configure()` and `Visualize()`, which must be implemented by any concrete processing class. `Configure()` method must return the content of task file, the list of files to upload to the server before calculations, and the list of files to download from the server after calculations. With `Visualize()` method, which is called by framework after downloading results from the server, a concrete processing operation performs result visualization. To execute a concrete processing operation, one must parameterize `executeParallelTask()` generic operation with this concrete operation class.

Also, the framework contains the tools for loading and visualizing images in various graphic formats.

### 3.3 Server Component

Server component is a UNIX application SSCCIP_Server implemented in C++ with the use of author's SSCC_PIPL library for high-performance image processing on multicomputer [3] briefly described below. SSCCIP_Server is MPI-based application, executing on each multiprocessor node by mpirun task launcher. SSCCIP_Server

source code is also organized as framework. Particularly, generic subprograms was created for some important types of processing procedures like pixel-to-pixel and neighborhood-to-pixel, implementing the actions which are common for all the procedures (parsing task file, loading source image with the support of various graphic formats, automatic determining parallelization parameters, parallelizing itself, storing result image in graphic file). To implement a concrete processing operation, one must create C++ algorithm class whose interface is compatible with SSCC_PIPL library.

### 3.4 SSCC_PIPL Library

SSCC_PIPL (Siberian Scientific Computing Center – Parallel Image Processing Library) library, computational core of SSCCIP system, is discussed in details in [3]. SSCC_PIPL is a C++ MPI-based library providing programmer with the environment for building efficient parallel image processing programs. Its features are:

1. SPMD (Single Program Multiple Data) computational model.
2. Reading/writing images in various graphic formats.
3. Hiding MPI parallel environment from the programmer.
4. Several representations for distributed image: storing image on one processor, replicating image on all processors, cutting image into horizontal strips, cutting image into horizontal strips with overlap.
5. Generic subprograms for pixel-to-pixel and neighborhood-to-pixel operations execution for all distributed image representations. The subprograms are parameterized with classes of concrete processing algorithms describing algorithms in terms of the neighborhood of the pixel being processed without touching parallel environment.
6. Low overhead of computational model abstraction (programs written with SSCC_PIPL library is just 10 percent slower than the ones written in pure MPI).
7. Great acceleration of whole process of parallel image processing program development (saving time for time-consuming designing, coding, and debugging MPI-based programs).

## 4 Example of Concrete Technology Implementation

Practical testing of SSCCIP framework was carried out via implementation of technology for circle structures detection in space images [4].

- The circle structures detection algorithm is a neighborhood-to-pixel operation; therefore the implementation of server part of the technology came to design of C++ class compatible with SSCC_PIPL library which describes the algorithm in terms of neighborhood of the pixel to be processed and subsequent parameterization of generic neighborhood-to-pixel SSCC_PIPL subroutine with this class.

- For the client part of the technology, it was necessary to implement procedures of algorithm configuration and result visualization.
  - Interactive configuration procedure allows operator to select paths to both source image and to processing result, radius of circle structures to search, statistical parameters of search procedure etc.; it was implemented via displaying dialog box.
  - The result of algorithm execution on multicomputer is an image with black background pixels and grey/white pixels marking the centers of the concave upward/downward circle structures. So the result visualization procedure was implemented by drawing black/white circles of given radius in original image, marking location of the concave upward/downward circle structures and subsequent visualizing this combined image.

All remaining operations (uploading source image and task file to multicomputer, enqueueing computational task and tracking its status, parsing task file, loading source image and storing result image, parallelizing calculations, and downloading result to the client computer) was implemented by SSCCIP framework code. As one can see, creating new high-performance technology did not require from programmer to deal with parallelism or network communications.

As was stated in [3], implementation of circle structure detection algorithm with SSCC_PIPL library results in just 10 percent performance overhead in comparison with pure MPI implementation which allows saying about efficiency of the whole technology developed in SSCCIP framework.

## 5 Conclusion

The main result of the work on SSCCIP software development is the creation of the framework for high-performance space data processing on remote multicomputer. The framework allows solving a various range of remote sensing data processing and analysis problems and a developer of concrete high-performance technology within the framework must implement only the details specific for the algorithm to be implemented.

The SSCCIP project is in its active development now which includes both common functionality extension and new concrete technologies addition. Currently the system allows simultaneous execution of several computational tasks on different multi computers and contains three remote sensing data processing technologies: beside the circle structures detection (mentioned above), these are the extraction of SAR-based texture features from aerospace images [5] and the mapping of space images to the digital map. Below is the screenshot of sample SSCCIP session: here, two parallel tasks (circle structures extraction and mapping space image to digital map) are completed and their results are visualized; operator is configuring the third task (SAR features extraction).

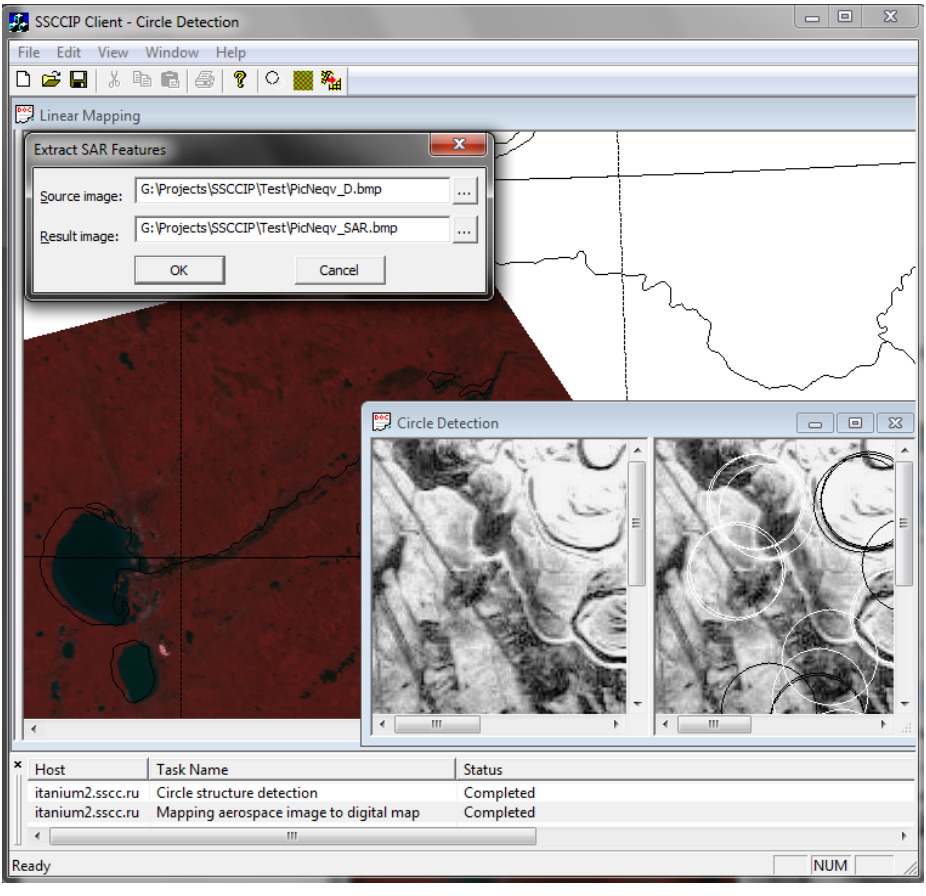



**Fig. 1.** Screenshot of typical SSCCIP session

## References

1. Plaza, A., Chang, C.-I.: High-Performance Computing in Remote Sensing. CRC Press, Boca Raton (2007)
2. CryptLib, `http://www.cs.auckland.ac.nz/~pgut001/cryptlib`
3. Rusin, E.V.: Object-Oriented Parallel Image Processing. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 344–349. Springer, Heidelberg (2009)
4. Alekseev, A.S., Pyatkin, V.P., Salov, G.I.: Crater Detection in Aerospace Imagery Using Simple Non-parametric Statistical Tests. In: Chetverikov, D., Kropatsch, W.G. (eds.) CAIP 1993. LNCS, vol. 719, pp. 793–799. Springer, Heidelberg (1993)
5. Sidorova, V.S.: Unsupervised Classification of Forest's Image by Texture Model Features. Pattern Recognition and Image Analysis 19(4), 698–703 (2009)

# Parallel Logical Cryptanalysis of the Generator A5/1 in BNB-Grid System

Alexander Semenov[1], Oleg Zaikin[1], Dmitry Bespalov[1], and Mikhail Posypkin[2]

[1] Institute for System Dynamics and Control Theory
Siberian Branch of Russian Academy of Sciences,
Lermontov str. 134, 664033 Irkutsk, Russia
[2] Institute for Systems Analysis of Russian Academy of Sciences,
Pr. 60-letiya Oktyabrya 9, 117312 Moscow, Russia
biclop@rambler.ru, oleg.zaikin@icc.ru, bespalov@altrixsoft.com,
posypkin@isa.ru

**Abstract.** In logical cryptanalysis a problem of search of a secret key of a cryptographic system is formulated as a SAT problem, i.e. a problem of search of a satisfying assignment for some CNF. In this paper we consider some natural strategies for parallelization of these SAT problems. We apply coarse-grained approach which makes it possible to use distributed computing environments with slow interconnect. The main practical result of this paper is successful logical cryptanalysis of keystream generator A5/1 in BNB-Grid system.

**Keywords:** Logical cryptanalysis, SAT, stream ciphers, A5/1, coarse-grained parallelization, Grid

## 1 Introduction

The idea of using SAT-solvers for the problems of cryptanalysis was first proposed in [1]. Term "logical cryptanalysis" itself was introduced in [2]. Examples of successful application of SAT approach to cryptanalysis of some weak stream ciphers are shown in [3], [4], [5]. However, to the best of our knowledge there are no results of successful use of parallel algorithms in logical cryptanalysis of widely used stream ciphers.

In this paper we consider the problem of parallel logical cryptanalysis of the stream generator A5/1 which is used to encrypt GSM-traffic. According to basic principles of logical cryptanalysis we reduce the problem of cryptanalysis of the generator A5/1 to a SAT-problem. Then we use special technique of parallelization to solve the obtained SAT-problem. This technique exploits peculiarities of the original SAT problem to decompose it into a large set of independent sub-problems. This approach was implemented in the BNB-Grid framework [6] specially developed for solving large scale problems in heterogeneous distributed systems. Using this approach we were able to successfully perform the cryptanalysis of the generator A5/1 in reasonable time.

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 473–483, 2011.

We also experimentally proved that the same keystream of an arbitrary length can be generated from different secret keys and identified all such keys for the particular 144-bit fragment of keystream.

## 2 Reducing Cryptanalysis of the Generator A5/1 to SAT

In this section we give general formulation of cryptanalysis problem for keystream generators and describe the procedure of reduction of this problem to SAT. Let $f_n$,

$$f_n : \{0,1\}^n \to \{0,1\}^*,$$

be a discrete function defined by the algorithm of the generator, that produces a keystream from a secret key $x \in \{0,1\}^n$. We consider the problem of cryptanalysis of keystream generator on the basis of a known keystream. The problem is to find the secret key using some fragment of a keystream and a known algorithm of its generation. It is easy to see that this problem is equivalent to the problem of inversion of the function $f_n$, i.e. the problem of finding such $x \in \{0,1\}^n$ that $f_n(x) = y$ if $y \in range\, f_n$ and an algorithm of computation of $f_n$ are known.

The first step of logical cryptanalysis consists in building a conjunctive normal form (CNF) encoding an algorithm of keystream generator. To obtain this CNF we use Tseitin transformations which were proposed by G.S. Tseitin in 1968 in [7]. In these transformations original function is usually represented by a Boolean circuit over an arbitrary complete basis, for example $\{\&, \neg\}$.

Let $f_n$ be a discrete function defined by an algorithm of generator. We will consider $f_n$ as a function of Boolean variables from the set $X = \{x_1, \ldots, x_n\}$. Let $S(f_n)$ be Boolean circuit which represents $f_n$ over $\{\&, \neg\}$. Each variable from $X$ corresponds to one of $n$ inputs of $S(f_n)$. For each logic gate $G$ some new auxiliary variable $v(G)$ is introduced. Every AND-gate $G$ is encoded by CNF-representation of Boolean function $v(G) \leftrightarrow u\&w$. Every NOT-gate $G$ is encoded by CNF-representation of Boolean function $v(G) \leftrightarrow \neg u$. Here $u$ and $w$ are variables corresponding to inputs of $G$. CNF encoding $S(f_n)$ is

$$\mathop{\&}_{G \in S(f_n)} C(G),$$

where $C(G)$ is CNF encoding gate $G$. Then

$$\left( \mathop{\&}_{G \in S(f_n)} C(G) \right) \cdot y_1^{\sigma_1} \cdot \ldots \cdot y_m^{\sigma_m}$$

is CNF encoding the inversion problem of the function $f_n$ in point $y = (\sigma_1, \ldots, \sigma_m)$. Here

$$y^\sigma = \begin{cases} \bar{y}, \text{if } \sigma = 0 \\ y, \text{if } \sigma = 1 \end{cases}$$

and $y_1, \ldots, y_m$ are variables corresponding to outputs of $S(f_n)$.

Quite often a structure of an algorithm calculating a cryptographic function allows us to write a system of Boolean equations which encodes this algorithm

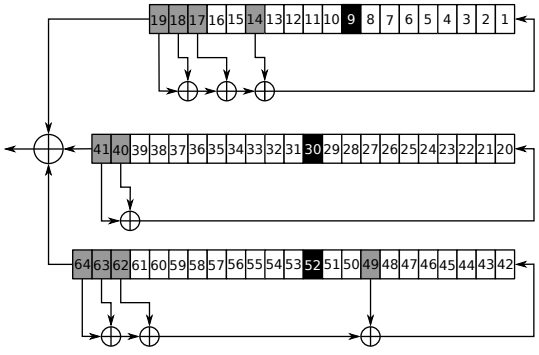



**Fig. 1.** Scheme of the generator A5/1

directly without constructing Boolean circuit. Using Tseitin transformations we can make a transition from the obtained system to one equation in the form "CNF=1".

Next, we consider the keystream generator A5/1 used to encrypt traffic in GSM networks. The algorithm of this generator became publicly available in 1999 after reverse engineering performed by M. Briceno. A lot of attacks on this cipher are described, however it is still actively used. The most recent attacks used technique of rainbow tables [8], however this approach can not guarantee the success in 100% of cases. Further we propose a new approach to cryptanalysis of the generator A5/1 that uses parallel algorithms for solving SAT problems.

The following description of the generator A5/1 (see Fig. 1) was taken from the paper [9]. According to [9] the generator A5/1 contains three linear feedback shift registers (LFSR, see, e.g., [10]), given by the following connection polynomials: LFSR 1: $X^{19}+X^{18}+X^{17}+X^{14}+1$; LFSR 2: $X^{22}+X^{21}+1$; LFSR 3: $X^{23}+X^{22}+X^{21}+X^{8}+1$.

The secret key of the generator A5/1 is the initial contents of LFSRs 1–3 (64 bits). In each unit of time $\tau \in \{1,2,\dots\}$ ($\tau=0$ is reserved for the initial state) two or three registers are shifted. The register with number $r$, $r \in \{1,2,3\}$, is shifted if $\chi_r^\tau\,(b_1^\tau, b_2^\tau, b_3^\tau)=1$, and is not shifted if $\chi_r^\tau\,(b_1^\tau, b_2^\tau, b_3^\tau)=0$. By $b_1^\tau,\ b_2^\tau,\ b_3^\tau$ we denote here the values of the clocking bits at the current unit of time. The clocking bits are 9-th, 30-th and 52-nd. Corresponding cells in Fig. 1 are black. The function $\chi_r^\tau\,(\cdot)$ is defined as follows

$$\chi_r^\tau\,(b_1^\tau, b_2^\tau, b_3^\tau)=\begin{cases}1, & b_r^\tau=\text{majority}\,(b_1^\tau, b_2^\tau, b_3^\tau)\\ 0, & b_r^\tau\neq\text{majority}\,(b_1^\tau, b_2^\tau, b_3^\tau)\end{cases}$$

where $\text{majority}\,(A,B,C)=A\cdot B\vee A\cdot C\vee B\cdot C$.

In each unit of time the values in the leftmost cells of the registers are added mod 2, the resulting bit is the bit of the keystream.

Thus, we can see that the generator A5/1 updates the content of each of the registers' cells as a result of conditional shifts: if the shift does not occur, then a new configuration of a register does not differ from the old one, otherwise values

of all cells of the register are updated. Hence with each cell at each unit of time we can associate a Boolean equation linking a new state of the cell with the previous one. Let variables $x_1, \ldots, x_{64}$ encode the secret key of the generator A5/1 ($x_i$ corresponds to cell with number $i \in \{1, \ldots, 64\}$). By $x_1^1, \ldots, x_{64}^1$ we denote variables encoding cells' state in the moment of time $\tau = 1$. System of equations which links these two sets of variables is:

$$\begin{cases} \left(x_1^1 \leftrightarrow x_1 \cdot \overline{\chi_1^1} \vee (\oplus_{i \in I} x_i) \cdot \chi_1^1\right) = 1 \\ \left(x_2^1 \leftrightarrow x_2 \cdot \overline{\chi_1^1} \vee x_1 \cdot \chi_1^1\right) = 1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \left(x_{20}^1 \leftrightarrow x_{20} \cdot \overline{\chi_2^1} \vee (\oplus_{j \in J} x_j) \cdot \chi_2^1\right) = 1 \\ \left(x_{21}^1 \leftrightarrow x_{21} \cdot \overline{\chi_2^1} \vee x_{20} \cdot \chi_2^1\right) = 1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \left(x_{42}^1 \leftrightarrow x_{42} \cdot \overline{\chi_3^1} \vee (\oplus_{k \in K} x_k) \cdot \chi_3^1\right) = 1 \\ \left(x_{43}^1 \leftrightarrow x_{43} \cdot \overline{\chi_3^1} \vee x_{42} \cdot \chi_3^1\right) = 1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \left(x_{64}^1 \leftrightarrow x_{64} \cdot \overline{\chi_3^1} \vee x_{63} \cdot \chi_3^1\right) = 1 \\ \left(g^1 \leftrightarrow x_{19}^1 \oplus x_{41}^1 \oplus x_{64}^1\right) = 1 \end{cases} \quad (1)$$

where $I = \{14, 17, 18, 19\}$, $J = \{40, 41\}$, $K = \{49, 62, 63, 64\}$ and $g^1$ is the first bit of keystream.

Let $g^1, \ldots, g^L$ be the first $L$ bits of the keystream of the generator A5/1. To the each bit $g^i, i \in \{1, \ldots, L\}$ we associate a system of the form (1). To find the secret key it is sufficient to find a common solution of these systems. The problem of finding of this common solution can be reduced by the means of Tseitin transformations to the problem of finding a satisfying assignment of a satisfiable CNF.

## 3 Coarse-Grained Parallelization of the Problem of Logical Cryptanalysis of the Generator A5/1

In this section we describe a technology for solving SAT problems in distributed computing systems (hereinafter DCS). Such systems consist of sets of computing nodes connected by a communication network. Each node of a DCS has one or several processors. Typical examples of DCS are computing clusters which have become widespread in recent years. The elementary computational units of modern DCS are cores of processors.

We consider an arbitrary CNF $C$ over the set of Boolean variables $X = \{x_1, \ldots, x_n\}$ and select in the set $X$ some subset

$$X' = \{x_{i_1}, \ldots, x_{i_d}\}\ ,\ \{i_1, \ldots, i_d\} \subseteq \{1, \ldots, n\}\,,$$

where $d \in \{1, \ldots, n\}$. We call $X' = \{x_{i_1}, \ldots, x_{i_d}\}$ a decomposition set and $d$ is the power of the decomposition set. To the decomposition set $X'$, $|X'| = d$,

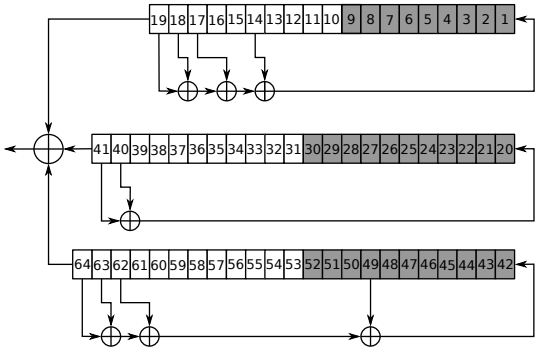



**Fig. 2.** Scheme of a decomposition set consisting of 31 variables

we associate the set $Y(X') = \{Y_1, \ldots, Y_K\}$ consisting from $K = 2^d$ different binary vectors of the length $d$, each of which is a vector of values of the variables $x_{i_1}, \ldots, x_{i_d}$. By $C_j = C|_{Y_j}$, $j = 1, \ldots, K$, we denote the CNF obtained after substitutions of the values from the vectors $Y_j$ to $C$. A decomposition family generated from the CNF $C$ by the set $X'$, is the set $\Delta_C(X')$, formed by the following CNFs:

$$\Delta_C(X') = \{C_1 = C|_{Y_1}, \ldots, C_K = C|_{Y_K}\}.$$

It is not difficult to see that any truth assignment $\alpha \in \{0,1\}^n$ satisfying $C$ ($C|_\alpha = 1$) coincides with some vector $Y^\alpha \in Y(X')$ in the components from $X'$ and coincides with some satisfying assignment of the CNF $C|_{Y^\alpha} \in \Delta_C(X')$ in the remaining components. In this case the CNF $C$ is unsatisfiable if and only if all the CNF in $\Delta_C(X')$ are unsatisfiable. Therefore, the SAT problem for the original CNF $C$ is reduced to $K$ SAT problems for CNFs from the set $\Delta_C(X')$. For processing the set $\Delta_C(X')$ as a parallel task list a DCS can be used.

We use peculiarities of original problem to construct a decomposition set with "good" properties. In logical cryptanalysis problems decomposition set is usually chosen among the subsets of the set of input variables of cryptographic function considered. For the logical cryptanalysis of A5/1 we propose to include into the decomposition set $X'$ the variables encoding the initial states of the cells of registers, starting with the first cells until the cells containing clocking bits inclusive (corresponding cells in the Fig. 2 are dark shaded). Thus, the decomposition set $X'$ consists of 31 variables:

$$X' = \{x_1, \ldots, x_9, x_{20}, \ldots, x_{30}, x_{42}, \ldots, x_{52}\} \tag{2}$$

This choice is motivated by the following considerations. Assigning values to all variables from $X'$ we determine the exact values of clocking bits for a large number of subsequent states of all three registers. These clocking bits are the most informative because they determine the value of the majority function.

Let $C$ be the CNF encoding the problem of cryptanalysis of the generator A5/1 (see Section 2). By $\Delta_{A5/1}(X')$ we denote the decomposition family generated from the CNF $C$ by the set $X'$ defined by (2). Thus $|\Delta_{A5/1}(X')| = 2^{31}$.

Further we describe a procedure of processing of $\Delta_{A5/1}(X')$ as a parallel task list in DCS. Let us put the CNFs of the family $\Delta_{A5/1}(X')$ in some order. We call an arbitrary CNF from $\Delta_{A5/1}(X')$ locked if at the current moment of time the SAT problem for it has either been solved or is being solved on some core of the DCS. The other CNFs are called free. We select first $M$ CNFs $C_1, \ldots, C_M$ from the family $\Delta_{A5/1}(X')$. For each of the selected CNFs we solve the SAT problem on a separate core of the DCS. Once some core is released we launch the procedure of solving of the SAT problem for the next free CNF of the family $\Delta_{A5/1}(X')$ on this core. This process continues until a satisfying assignment for some CNF from $\Delta_{A5/1}(X')$ is found, or until the unsatisfiability of all CNFs from $\Delta_{A5/1}(X')$ is proven.

## 4 Modification of a SAT Solver for Solving the Problem of Logical Cryptanalysis of the Generator A5/1

For solving of SAT problems from the decomposition family $\Delta_{A5/1}(X')$ we used a modified version of well-known SAT solver MiniSat-C_v1.14.1 [11]. The first stage of the modification consists in changing the decision variable selection procedure (see [12]) implemented in Minisat. Namely, a procedure of assignment of initial activity (different from zero) for those variables in the CNF which correspond to the input variables of the function was added. For the problems of cryptanalysis of generators this method allows to select, on the initial stage of the solving process, the variables corresponding to the secret key as priority variables for decision variable selection procedure. Also some basic constants of the solver were changed. Like most of its analogs Minisat periodically changes the activity of all the variables and clauses in order to increase the priority of selection for variables from the clauses derived in the later steps of the search. Moreover, in 2% of cases the Minisat assigns a value to a variable selected randomly, rather than to the variable with the maximum activity. These heuristics show, on average, good results on a broad set of test examples used in the competitions of SAT solvers. However, for the CNFs encoding problems of cryptanalysis they are, usually, not efficient. In all our experiments we use modified SAT solver Minisat-C v1.14.1 in which periodical lowering of the activity and random selection of variables are prohibited. In total, these simple changes led to a substantial increase in efficiency of the SAT solver on cryptographic tests. Unmodified SAT solvers Minisat-C v1.14.1 and Minisat 2.0 did not cope with CNFs from the decomposition family constructed during the logical cryptanalysis of the generator A5/1, even in 10 minutes of work (the computations were interrupted). The modified Minisat-C v1.14.1 solved these problems in less than 0.2 seconds on average.

In the preceding section a general procedure for parallel processing of a list of tasks was described. During this procedure the control process monitors the load of computing cores and sends new tasks to the released cores. In practice,

a direct implementation of this scheme leads to an excessive growth of transfer costs, but provides uniform load of the cores.

The efficiency of a SAT solver in a DCS can be improved by using job batches. Each job batch is a subset of the decomposition family $\Delta_{A5/1}(X')$. Sending batches instead of single CNFs allows to reduce the cost of the transfer. We decompose $\Delta_{A5/1}(X')$ into set of disjoint job batches. The obtained set is considered as a task list where each job batch is a list item. For processing this task list we use the technique described in the previous section.

The fact that a decomposition set is a set of Boolean variables makes the problem of transferring the batches to the cores very simple. Indeed, let $X'$ be decomposition set defined by (2). And let $M$ be the number of computing cores in the DCS. The core with the number $p \in \{1,\ldots,M\}$ we denote by $e_p$. For the sake of simplicity, assume that $M = 2^k$, $k \in \mathrm{N}_1$, and $k < 31$. If we suppose that all the tasks in the decomposition family $\Delta_{A5/1}(X')$ have approximately equal complexity, then when solving the problem in the DCS each core is going to process approximately the same number of tasks. This means that the decomposition family $\Delta_{A5/1}(X')$ can be partitioned into $2^k$ subfamilies of equal power and each subfamily can be further processed entirely on the corresponding core. For this purpose select in $X'$ some subset $X'_k$ of power $k$ ($X'_k$ can be formed, for example, by the first $k$ variables from $X'$). The description of the job batch for a particular $e_p, p \in \{1,\ldots,2^k\}$, is a binary vector $\alpha_p$ of the length $k$, formed by the values of variables from $X'_k$. Next, for each $e_p, p = 1,\ldots,2^k$, we consider the set $\Lambda_p$, consisting of $2^{31-k}$ different vectors of the length 31 of the form $(\alpha_p|\beta)$, where $\beta$ takes all $2^{31-k}$ possible values from the set $\{0,1\}^{31-k}$.

Each core $e_p, p \in \{1,\ldots,2^k\}$, receives its job batch from the control process as a vector $\alpha_p$ which is used for constructing the set $\Lambda_p$. A subfamily of the family $\Delta_{A5/1}(X')$ processed by $e_p$ is obtained as a result of substituting vectors from $\Lambda_p$ to CNF $C$ which encodes the problem of cryptanalysis of the generator A5/1.

## 5 Implementation of Parallel Logical Cryptanalysis of the Generator A5/1 in BNB-Grid System

Using the decomposition set $X'$ defined by (2) we calculated an approximate time of parallel logical cryptanalysis of the generator A5/1 on the "Chebyshev" cluster [13].

In our experiments for several variants of a keystream length we constructed random samples of the volume of 1000 CNFs. For each sample we solved all SAT problems from this sample and calculated average time of their solving. For this purpose we used one core of Intel E8400 processor. In the Table 1 we show the approximate time of solving the problem of logical cryptanalysis of the generator A5/1 on one core of this processor for different variants of keystream lengths. On the basis of these results we decided to use for cryptanalysis of the generator A5/1 first 144 bits of keystream.

**Table 1.** Approximate time of logical cryptanalysis for the generator A5/1 on a single core of the processor Intel E8400 (in hundreds millions of seconds)

| **Keystream length** | 128 | 144 | 160 | 176 | 192 |
|---|---|---|---|---|---|
| **Approximate time** | 3.76 | **3.55** | 3.71 | 3.73 | 3.81 |

**Table 2.** Characteristics of processors

| **Processor model** | Intel E8400 | Intel E5472 |
|---|---|---|
| **Number of cores** | 2 | 4 |
| **Core frequency** | 3.0 GHz | 3.0 GHz |
| **Bus frequency** | 1333 MHz | 1600 MHz |
| **Cache L2** | 6 Mb | 12 Mb |

From the Table 2 we can see that the cores of Intel E8400 and Intel E5472 processors are comparable in power (there is only a slight difference in the bus frequency). Cluster "Chebyshev" [13] is based on Intel E5472 processors, therefore, using the results of our experiments on Intel E8400 processor we can calculate an approximate time required for solving the problem of parallel logical cryptanalysis of the generator A5/1 on cluster "Chebyshev".

According to Table 1 parallel logical cryptanalysis of the generator A5/1 would take about one day of "Chebyshev" work even if the cluster is fully dedicated to this task. However, exclusive use of publically available supercomputers is usually not possible. Thus it was clear that for a successful solving of cryptanalysis of the generator A5/1 we would need to combine computational powers of several supercomputers.

We decided to use the BNB-Grid [6] which is a generic framework for implementing some combinatorial algorithms on distributed systems. The BNB-Grid tool can harness the consolidated power of computing elements collected from service Grids, desktop Grids and standalone resources. Adding different types of computational resources is available (e.g., Unicore service Grid, BOINC desktop Grid system). This package has already proved its efficiency in solving several large scale optimization problems [6,14].

Hierarchical structure of BNB-Grid is shown on the Fig. 3. On the top level of the BNB-Grid the object Computing Space Manager (CS-Manager) is located. It decomposes the original problem into subproblems and distributes them among the computing nodes. For each computing node there is a corresponding object of the type Computing Element Manager (CE-Manager). CE-Manager provides communication between CS-Manager and the corresponding computing node and also starts and stops applications on this node. After receiving a task from the CS-Manager, CE-Manager transfers it to the corresponding node and starts MPI application BNB-solver which processes the received task on all available cores.

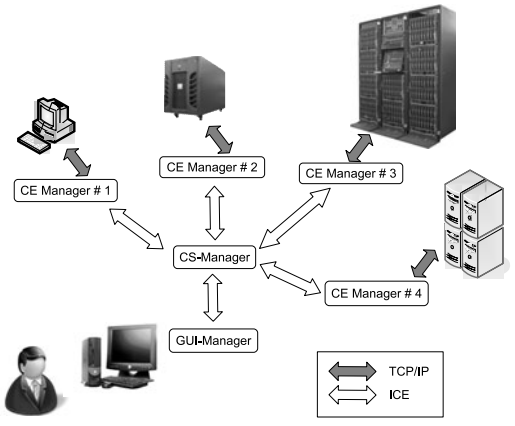



**Fig. 3.** Organization of computations in BNB-Grid

A module for processing SAT problems on a computing cluster was added to the BNB-Solver. The input data of the control object CS-Manager is a description of the original SAT problem in XML format. CS-Manager decomposes SAT problem for the original CNF $C$ and obtains decomposition family. For transferring tasks between the CS-Manager and CE-Managers the technique of job batches described in Section 4 is used. Each job batch is a compact description of a subset of the decomposition family. Sending of batches instead of single CNFs allows to reduce the cost of the transfer.

The computations were carried out on a distributed system consisting of four computing clusters (see [15]): MVS-100k (Joint Supercomputer Center of RAS), SKIF-MSU "Chebyshev" (Moscow State University), cluster of RRC Kurchatov Institute, BlueGene P (Moscow State University).

In our experiments three test problems of cryptanalysis of the generator A5/1 were solved. During the computational experiment the number of simultaneously working computing cores varied from 0 to 5568, averaging approximately 2–3 thousand cores. For each test the computations stopped after finding the first satisfying assignment. The first test problem was solved (the secret key of the generator was found) in 56 hours, the second and the third—in 25 and 122 hours respectively.

The problem of cryptanalysis of the generator A5/1 is also interesting because the same keystream of arbitrary length can be generated from different secret keys. This fact was noted by J. Golic in [16]. We denote these situations as "collisions" using the evident analogy with the corresponding notion from the theory of hash functions. The approach presented in the paper allows us to solve the problem of finding all the collisions of the generator A5/1 for a given fragment of a keystream. Using BNB-Grid all collisions for one test problem (we analyzed the first 144 bits of keystream) were found. It turned out that there are only three such collisions (see Table 3). Processing this test problem by the distributed system described above took 16 days.

**Table 3.** Original key and collisions of the generator A5/1 (in hexadecimal format)

| | **LFSR 1** $x_1, \ldots, x_{19}$ | **LFSR 2** $x_{20}, \ldots, x_{41}$ | **LFSR 3** $x_{42}, \ldots, x_{64}$ |
|---|---|---|---|
| **original key** | 2C1A7 | 3D35B9 | EEAF2 |
| **collision** | 2C1A7 | 3E9ADC | EEAF2 |
| **collision** | 2C1A7 | 3D35B9 | 77579 |

## 6 Conclusion and Future Work

In this paper the results of successful parallel logical cryptanalysis of the generator A5/1 are presented. For solving this problem a Grid-system was specially constructed. Although the solving of the considered cryptanalysis problem turned out to be quite time-consuming (from 1 to 16 days), all the tests were correctly solved. Thus, the possibility of cryptanalysis of A5/1 in publicly available distributed computing systems (e.g., BOINC, [17]) was experimentally confirmed. It should be noted that our approach does not require any special hardware (like, for example, in [8]). On the webpage [18] CNFs encoding the problem of cryptanalysis of the generator A5/1 are available.

We believe that logical cryptanalysis problems form quite a perspective class of tests for new technologies of solving combinatorial problems in distributed computing environments. In the nearest future we plan to apply the proposed approach to solving problems of cryptanalysis of some stream ciphers and hash functions.

**Acknowledgements.** The authors would like to thank Stepan Kochemazov, Alexey Hmelnov and Alexey Ignatiev (ISDCT SB RAS) for their help and numerous valuable advices. This work is supported by Russian Foundation for Basic Research (Grants No. 11-07-00377-a and No. 10-07-00301-a) and Lavrentiev grant of SB RAS.

## References

1. Cook, S.A., Mitchel, D.G.: Finding hard instances of the satisfiability problem: A survey. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, pp. 1–17 (1997)
2. Massacci, F., Marraro, L.: Logical Cryptanalysis as a SAT Problem. Journal of Automated Reasoning 24(1-2), 165–203 (2000)
3. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with Minisat. Technical Report, 2007/040, ECRYPT Stream Cipher Project (2007)
4. Semenov, A.A., Zaikin, O.S., Bespalov, D.V., Ushakov, A.A.: SAT-approach for cryptanalysis of some stream ciphering systems. Journal of Computational Technologies 13(6), 134–150 (2008) (in Russian)

5. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT Solvers to Cryptographic Problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009)
6. Afanasiev, A., Posypkin, M., Sigal, I.: Project BNB-Grid: solving large scale optimization problems in a distributed environment. In: 21 International Symposium on Nuclear Electronics and Computing, Dubna, pp. 15–19 (2008)
7. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic, part 2, pp. 115–125 (1968)
8. Guneysu, T., Kasper, T., Novotny, M., Paar, C., Rupp, A.: Cryptanalysis with COPACOBANA. IEEE Transactions on Computers 57(11), 1498–1513 (2008)
9. Biryukov, A., Shamir, A., Wagner, D.: Real time cryptanalysis of A5/1 on a PC. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 1–18. Springer, Heidelberg (2001)
10. Menezes, A., Van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
11. The MiniSat page, http://www.minisat.se
12. Marqeus-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. on Computers 48(5), 506–521 (1999)
13. Chebyshev supercomputer, http://parallel.ru/cluster/skif_msu.html
14. Evtushenko, Y., Posypkin, M., Sigal, I.: A framework for parallel large-scale global optimization. Computer Science – Research and Development 23(3), 211–215 (2009)
15. Top 50 CIS Supercomputers, http://www.supercomputers.ru
16. Golic, J.: Cryptanalysis of Alleged A5 Stream Cipher. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 239–255. Springer, Heidelberg (1997)
17. BOINC: Open-source software for volunteer computing and grid computing, http://boinc.berkeley.edu/
18. Parallel logical cryptanalysis of the generator A5/1, http://sat.all24.org

# High Performance Computing of MSSG with Ultra High Resolution

Keiko Takahashi[1], Koji Goto[2], Hiromitsu Fuchigami[3], Ryo Onishi[1], Yuya Baba[1], Shinichiro Kida[1], and Takeshi Sugimura[1]

[1] The Earth Simulator Center, JAMSTC, 3173-25 Showa-machi, Kanazawa-ku, Yokohama, 236-0001, Japan
[2] NEC Corporation, 1-10 Nisshin-cho, Fuchu-shi, Tokyo, 183-5801, Japan
[3] NEC Informatec Systems LTD, 3-2-1 Sakato, Takatsu-ku, Kawasaki-shi, Kanagawa, 213-0012, Japan
takahasi@jamstec.go.jp

**Abstract.** Multi-Scale Simulator for the Geoenvironment (MSSG), which is a coupled non-hydrostatic atmosphere-ocean-land model, has been developed in the Earth simulator Center. Outline of MSSG is introduced and its characteristics are presented. In MSSG, Yin-Yang grid system is adopted in order to relax Courant–Friedrichs–Lewy condition on the sphere. Furthermore, the Large-Eddy Simulation model for the turbulent atmospheric boundary layer and cloud micro physics model have been adapted for ultra high resolution simulations of weather/climate system. MSSG was optimized computationally on the Earth Simulator and its dynamical core processes had attained 51.5 Tflops on the Earth Simulator. Results from preliminary validations including forecasting experiments are presented.

**Keywords:** Coupled atmosphere-ocean model, Earth Simulator, High performance computing, Multi-scale simulations.

## 1 Introduction

Intense research effort is focused on understanding the climate/weather system using coupled atmosphere-ocean models. It is widely accepted that the most powerful tools available for assessing future weather/climate are fully coupled general circulation models. Not only interactions between atmosphere or ocean components, but also various components have been coupled with various interactive ways and influence on earth system. Getting further information on perspectives of future weather/climate and the earth system, whole of the earth system should be simulated using coupled models as much as we can.

The Earth Simulator Center have been developed coupled non-hydrostatic atmosphere-ocean-land general circulation model, which is called Multi-Scale Simulator for the Geoenvironment (MSSG), to be run on the Earth Simulator with ultra high resolution and really high performance computing architectures. When development of the coupled model has been completed and various simulations are capable on the Earth Simulator, the ambition task of simulating and understanding the earth system should bring us further detail information on the Earth System. Those

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 484–498, 2011.

high qualified information for forecasting or objecting, more significant impacts might be brought by the results of simulations. Simulations with target scale shown in Fig.1 will be planed in near future.

In this paper, MSSG is described in section 2. In section 3, implementation architectures of MSSG on the Earth Simulator are introduced. Performance analysis is performed and results are presented in section 4. Preliminary validation results from simulation with MSSG are shown in section 5.

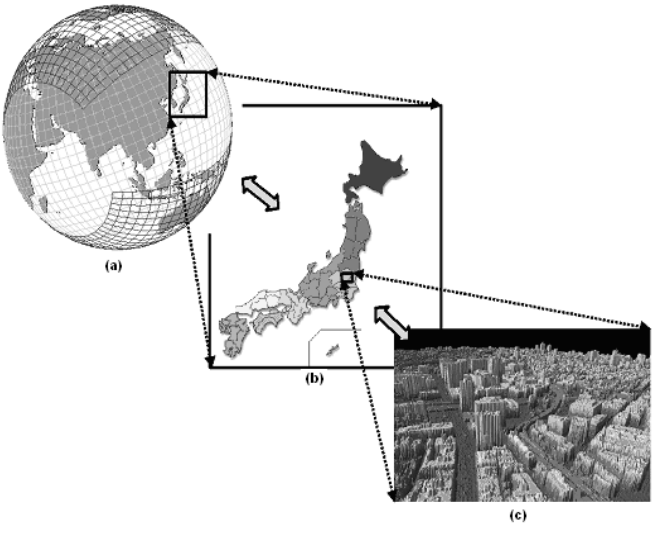



**Fig. 1.** (a): Yin-Yang gird system for the global. Each colored panel is corresponding to Yin and Yang grid, respectively. (b): Japan region is nested with two way interaction to the global. (c): As our near future target simulations, urban scale weather/climate simulations will be allowed with two-way interactions to the global/regional scale simulations. Urban topography was lent by Geographical Survey Institute.

## 2 Model Description

### 2.1 The Atmospheric Component: MSSG-A

The atmosphere componentof MSSG, MSSG-A is compromised of the non-hydrostatic, fully compressive flux form of dynamic[1] and Smagorinsky-Lilly type parameterizations [2][3] for subgrid scale mixing, surface fluxes [4][5], cloud microphysics with mixed phases[6], cumulus convective processes [7][8] and simple radiation scheme. The set of the prognostic equations is presented as follows:

In equations (1)-(7), prognostic valuables are momentum $\rho\mathbf{v} = (\rho u, \rho v, \rho\omega)$, $\rho'$ which is calculated as $\rho' = \rho - \bar{\rho}$ and P' defined by $P' = P - \bar{P}$. $\rho$ is the density; P is the pressure; $\bar{P}$ is a constant reference pressure. $\mathbf{f}, \mu, \kappa,$ and $\gamma$ are the Coriolis force, the viscosity coefficient, the diffusion coefficient, and the ratio of specific heat, respectively. F is the heat source term and the viscosity term, respectively. G is the metric term for vertical coordinate; λ is latitude; φ is longitude.

$$\frac{\partial \rho'}{\partial t} + \frac{1}{G^{\frac{1}{2}} a \cos\varphi} \frac{\partial (G^{\frac{1}{2}} G^{13} \rho u)}{\partial \lambda} + \frac{1}{G^{\frac{1}{2}} a \cos\varphi} \frac{\partial (G^{\frac{1}{2}} G^{23} \cos\varphi \rho v)}{\partial \varphi} + \frac{1}{G^{\frac{1}{2}}} \frac{\partial (\rho w^*)}{\partial z^*} = 0, \tag{1}$$

$$\frac{\partial \rho u}{\partial t} + \frac{1}{G^{\frac{1}{2}} a \cos\varphi} \frac{\partial (G^{\frac{1}{2}} G^{13} P')}{\partial \lambda} = -\nabla \bullet (\rho u \mathbf{v}) + 2 f_r \rho v - 2 f_\varphi \rho w + \frac{\rho v u \tan\varphi}{a} - \frac{\rho w u}{a} + F_\lambda, \tag{2}$$

$$\frac{\partial \rho v}{\partial t} + \frac{1}{G^{\frac{1}{2}} a} \frac{\partial (G^{\frac{1}{2}} G^{23} P')}{\partial \varphi} = -\nabla \bullet (\rho v \mathbf{v}) + 2 f_\lambda \rho w - 2 f_r \rho u - \frac{\rho u u \tan\varphi}{a} - \frac{\rho w v}{a} + F_\varphi, \tag{3}$$

$$\frac{\partial \rho w}{\partial t} + \frac{1}{G^{\frac{1}{2}}} \frac{\partial P'}{\partial z^*} + \rho' \mathbf{g} = -\nabla \bullet (\rho w \mathbf{v}) + 2 f_\varphi \rho u - 2 f_\lambda \rho v + \frac{\rho u u}{a} + \frac{\rho v v}{a} + F_r, \tag{4}$$

$$\frac{\partial P'}{\partial t} + \nabla \bullet (P \mathbf{v}) + (\gamma - 1) P \nabla \bullet \mathbf{v} = (\gamma - 1) \nabla \bullet (\kappa \nabla T) + (\gamma - 1) \phi, \tag{5}$$

$$P = \rho R T, \tag{6}$$

$$\rho w^* = \frac{1}{G^{\frac{1}{2}}} \left( G^{\frac{1}{2}} G^{13} \rho u + G^{\frac{1}{2}} G^{23} \rho v + \rho w \right). \tag{7}$$

Over land, the ground temperature and ground moisture are computed by using a bucket model as a simplified land model. As upper boundary condition, Rayleigh friction layer is set. For the lateral boundary condition of regional version, sponge type boundary condition [9] is used.

Regional version of the MSSG-A is utilized with one way nesting scheme by choosing the target region on the sphere, although two-way nesting is available as an option. Any large regions can be selected from the global, because both Coriolis and metric terms are introduced in the regional formulation. As another option, multiple regions are allows to be selected at the same time and computed with parallel.

### 2.2 The Ocean Component: MSSG-O

In the ocean component. MSSG-O, the in-compressive and hydrostatic equations with the Boussinesq approximation are used based on describing in [10][11]. The set of equations in the ocean component becomes as follows,

$$\frac{\partial c}{\partial t} = -\mathbf{v}gradc + F_c \tag{8}$$

$$\frac{\partial T}{\partial t} = -\mathbf{v}gradT + F_T \tag{9}$$

$$0 = \nabla \bullet \mathbf{v} = \left( \frac{1}{r\cos\varphi}\frac{\partial u}{\partial \lambda} + \frac{1}{r\cos\varphi}\frac{\partial(\cos\varphi v)}{\partial \varphi} + \frac{1}{r^2}\frac{\partial(r^2 w)}{\partial r} \right) \tag{10}$$

$$\frac{\partial u}{\partial t} = -\mathbf{v}gradu + 2f_{\mathrm{r}}v - 2f_{\varphi}w + \frac{vu\tan\varphi}{r} - \frac{wu}{r} - \frac{1}{\rho_0 r\cos\varphi}\frac{\partial P'}{\partial \lambda} + F_{\lambda} \tag{11}$$

$$\frac{\partial v}{\partial t} = -\mathbf{v}gradv + 2f_{\lambda}w - 2f_{r}u - \frac{uu\tan\varphi}{r} - \frac{wv}{r} - \frac{1}{\rho_0 r}\frac{\partial P'}{\partial \varphi} = +F_{\varphi} \tag{12}$$

$$\frac{\partial w}{\partial t} = -\mathbf{v}gradw + 2f_{\varphi}u - 2f_{\lambda}v + \frac{uu}{r} + \frac{vv}{r} - \frac{1}{\rho_0}\frac{\partial P'}{\partial r} - \frac{\rho'}{\rho_0}\mathbf{g} + F_r \tag{13}$$

$$\rho = \rho(T, c, P_0) \tag{14}$$

$$\frac{d}{dr}P_0 = -\rho_0 g(r) \tag{15}$$

where the Boussinesq approximation is adopted in (9) and all variables are defined as above for the atmospheric component. In equation (14), UNESCO scheme [12] is used.

Smagorinsky type scheme [2][3] is used as the subgrid-scale mixing in identical experiments with the ocean component. The level-2 turbulence closure of Mellor Yamada [13] has been also introduced to the ocean component as one of optional schemes.

In the ocean component, sponge layers are used for lateral boundary in the open ocean. The lateral boundary condition between ocean and land is defined as $\partial T/\partial t = \partial S/\partial t = 0$ and $\mathbf{v} = 0$. Bottom boundary condition is defined by Neumann condition without vertical velocity. The upper boundary conditions are given as momentum fluxes by wind, heat and fresh water fluxes from observational data of atmosphere.

### 2.3 Grid Configuration

Yin-Yang grid system [14] is used both for the MSSG-A and MSSG-O. Yin-Yang gird system as shown Fig.1(a) is characterized by overlapped three dimensional two

panels to cover the sphere. Basically, one component grid is defined as a part of low-latitude region covered between 45N and 45S and 270 in longitude of the usual latitude-longitude grid system and the other component of the grid system is defined in the same way but in different spherical coordinates. The region covered by a panel is able to change by rotating axes of the panels.

By using Yin-Yang grid system, we can find a solution on an issue of how to avoid singular points such as the south and north poles on a latitude/longitude grid system. In addition, the advantage to enlarge the time step is compared to conventionally utilized latitude/longitude grid system.

### 2.4 Differencing Schemes

In both the MSSG-A and MSSG-O, the Arakawa C grid is used. The MSSG-A utilizes the terrain following vertical coordinate with Lorenz type variables distribution [15]. The MSSG-O uses the z-coordinate system for the vertical direction. In discritization of time, the 2nd, 3rd and 4th Runge-Kutta schemes and leap-flog schemes with Robert-Asselin time filter are available. The 3rd Runge-Kutta schemes is adopted for MSSG-A. In this study, leap-flog schemes with Robert-Asselin time filter is used for MSSG-O.

For momentum and tracer advection computations, several discritization schemes are available [16][17]. In this study, the 5th order upwind scheme is used for MSSG-A and central difference is utilized in MSSG-O.The vertical speed of sound in the atmosphere is dominant comparing horizontal speed, because vertical discritization is tend to be finer than horizontal discritization. From those reasons, Horizontally explicit vertical implicit (HEVI) schemeis adopted in the atmosphere component. The speed of sound in the ocean is three times faster than it in the atmosphere, implicit method is introduced and Poisson equation (16) is solved in the method. Poisson equation is described as follows. Here is solved under Neumann boundary condition of $n \bullet gradP = n \bullet \mathbf{G}_{\mathrm{v}}$.

No side-effects of over lapped grid system such as Yin-Yang grid were considered due to validations results of various benchmark experiments[16][18][19][20][21].

$$\nabla \bullet gradP' = B, \tag{16}$$

$$B = \rho_0 \nabla \bullet \mathbf{G}_{\mathrm{v}} = \frac{\rho_0}{r\cos\varphi}\frac{\partial}{\partial\lambda}G_u + \frac{\rho_0}{r\cos\varphi}\frac{\partial}{\partial\varphi}(\cos\varphi G_v) + \frac{\rho_0}{r^2}\frac{\partial}{\partial r}(r^2 G_w), \tag{17}$$

### 2.5 Algebraic Multigrid Method in a Poisson Solver

Algebraic Multi-Grid (AMG) method [22] is used in order to solve the Poisson equation mentioned in section 2.4. AMG is well known as an optimal solution method. We used the AMG library which has been developed by Fuji Research Institute Corporation. The AMG library is characterized in terms of following points,

- AGM in the library has been developed based on aggregation-type AMG [22].
- In the library, AMG is used as a pre-conditioner in Krylov subspace algorithms.

- Incomplete LU decomposition (ILU) is adopted as a smoother in the library, which shows good computational performance even for ill-structured matrixes
- Local ILU is used for parallelization, in addition, fast convergence speed has been kept.
- Aggregation without smoothing is adopted with recalling procedure, because remarkably fast convergence has been performed by using the aggregation.

# 3 Implementation of MSSG on the Earth Simulator

## 3.1 Coding Style

MSSG is composed of non-hydrostatic atmosphere, hydrostatic/non-hydrostatic ocean and simplified land components with 200, 000 lines of the code. It is written in Fortran 90 and automatic allocation schemes are used in order to save memory size. MODULE features are used to keep maintainability and readability of the code. Extensions to the Message Passing Interface (MPI-2) has been used, which is tuned up for the scalability of the Earth Simulator.

## 3.2 Distribution Architecture and Communications

The architecture and data structures are based on domain decomposition methods. In Yin-Yang grid system, communication cost imbalance might occur by adopting one dimensional decomposition. The case of decomposition with 16 processes is considered in Fig.2. Each gray color is corresponding to each process. The number of arrows linking between different colored areas is corresponding to a mount of communication between processes. For example, in Fig.2 (a) for one dimensional domain decomposition, black colored process called A should communicate to different colored 8 processes. In Fig.2 (b) for two dimensional decomposition, a black colored process called A communicates two processes. In Fig.2 (a), communication data size is small, in addition, the number of communications is increased. When the same number for decomposition is defined in both (a) and (b), it is clear that less amount of communication realizes in Fig.2 (b). Two dimensional decomposition was adopted for both MSSG-A and MSSG-O due to the reasons mentioned above.

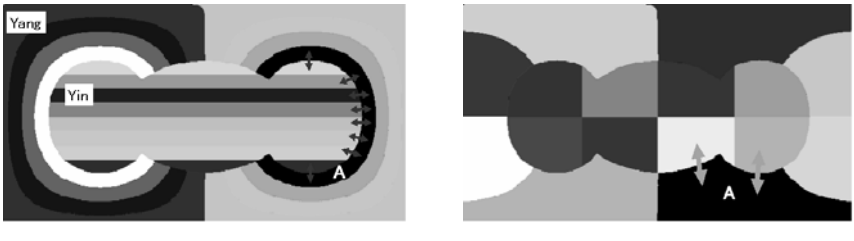



**Fig. 2.** Schematic features of domain decomposition on Yin-Yang grid system. Left: one dimensional domain decomposition, and Right: two dimensional domain decomposition.

### 3.3 Inter-/Intra-node Parallel Architectures and Vector Processing

Since effective parallelization and vectorization contribute to achieve high performance, the three-level parallelism which are inter-node parallel processing for distributed memory architecture, and intra-node parallel processing for shared memory architecture, and vector processing for a single processor should be utilized in order to pursuit high computational performance. MPI-based parallelism for inter-node is used to communicate among decomposed domains.

Micro-tasking of intra-node operations for shared memory architecture contributes to significant high performance, when long vector length is selected for the parallelism at a DO loop level. In order to equally share the computational load by 8 threads of micro-tasking, it is simple way that each micro-task is mapped onto a processor and vertical layers and latitudinal grid points are parallelized with micro-tasking architectures. Therefore, it is necessary to high computational performance that the number of vertical layers and latitudinal grid points are required to be a multiplier of 8 to archive higher performance computation. In the cases in this paper, 32 and 40 vertical layers have been selected for the MSSG-A and the MSSG-O, respectively.

When domain decomposition is used for inter-node parallelization, vector length and length of DO loops should be taken into account to be fully utilized. In this paper, two approaches in DO loops are considered in order to keep the length of DO loops. The first step to keep the length of DO loops is that both latitude and longitude direction are selected as an axis of DO loops. When the first approach is chosen, double looping structure is adopted. The second approach is that single DO looping structure is used by combining both looping axes of longitude and latitude direction. Fig.3 shows preliminary comparison results from computations of dynamical core with double looping and single looping structures. When single looping structure is adopted, array structures should be taken in account order to access grid point in overlapped regions of Yin-Yang grid system. In Fig.3, 1loop_list, 1loop_nolist and 1loop_metric present implementation architectures with list structure, without list structure and list structure excepting metric terms, respectively. 2loops shows results of cost performance with double DO looping structures. Single DO looping structure without list structures to access grid points shows best performance as shown in Fig.3. However, increasing length of DO loop, the discrepancy between double and single DO looping structure is getting small. Ultra high resolutions over 400 of loop length, which is corresponding to higher resolution than 25km for global, are required. Therefore, we adopted double DO looping structure, because fully high performance is expected as the same level performance of single DO looping structure and simplified coding styles are able to use in double DO looping structure.

In terms of above considering, two dimensional decomposition, computational performance of advection terms in dynamical core is presented in Fig.4. As vector length is selected near a multiplier of 256, computational performance shows well.

### 3.4 Memory and Ccost Reductions for Land Area in MSSG-O

By using two dimensional domain decomposition, computations over land are not needed in the MSSG-O. Generally, in one dimensional domain decomposition,

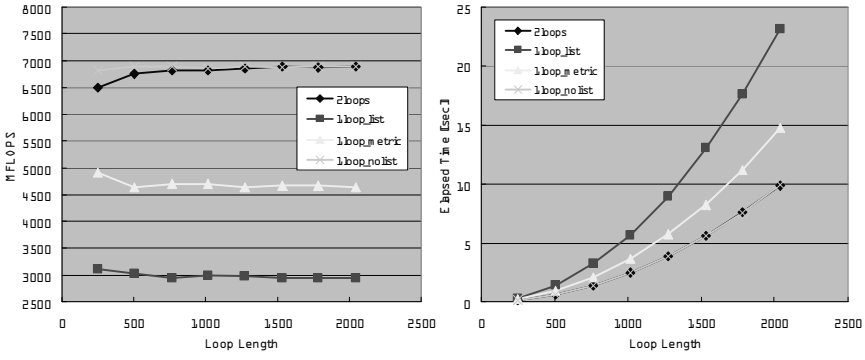



**Fig. 3.** Cost performance for double and single DO loop structures with different horizontal resolutions. Left and right figures show Mflops and elapsed time increasing resolutions.

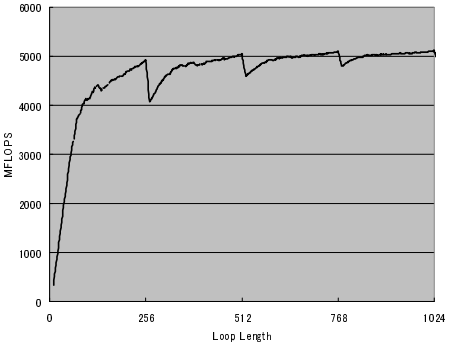



**Fig. 4.** Cost performance of dominant computations for advection terms in the dynamical core, increasing vector length

masking operation is used for computations in land area. Since cost of computation for land area accounts for about 22 %, those computation should be considered in order to reduce both memory and computational cost. In this paper, nodes allocated for computation in only land area are eliminated in advance of simulations. 22% of memory and of cost performance have been reduced by excluding redundant computations.

### 3.5 Overlapped Computations in MSSG-O

In the MSSG-O, procedures of AMG, advection computations of temperature and salinity, and computation of density are dominant in simulations of the ocean component. Since procedures of AMG are able to be performed independently of advection and density computations. Micro-tasking of intra-node operations for shared memory architecture is used for parallelization of AMG procedures and advection computations. Fig.5 shows an outline of parallelization with micro-tasking architecture. After using eight micro-tasking for computations of velocities u and v, one micro-task is used for Poisson solver with AMGCG and seven micro-tasks are

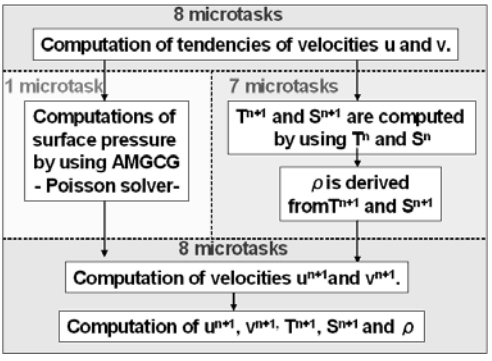



**Fig. 5.** Schematic figure of flow chart for parallelization with maicrotasking architectures in the ocean component

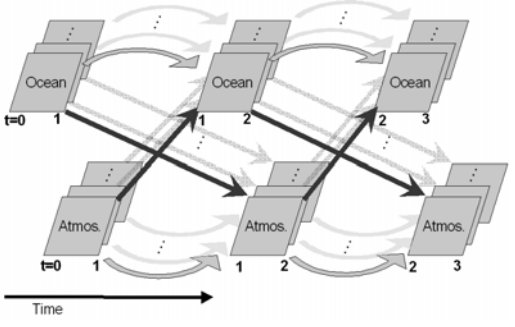



**Fig. 6.** Schematic figure of serial coupling scheme between the atmosphere and ocean components. Arrows show the direction of data flow.

used for computations of temperature, salinity and density. After parallel computation with micro-tasking architectures, velocities are computed and communication has performed by using full micro-taskings.

### 3.6 Coupling Scheme with High Computational Performance

The interface between atmosphere and ocean should be taken into account to maintain a self consistent representation in a model. The heat fluxes, moisture and momentum fluxes computed and averaged in MSSG-A transfer to MSSG-O as the upper boundary condition of it. Sea surface temperature (SST) integrated in MSSG-O transfers to the MSSG-A as bottom boundary condition. To remove time inconsistency through coupling, two approaches are considered. The one of them is that individual component can run independently during the same duration and communicate surface variables at the same time after integrating. In this framework, it is required to find the most suitable number of nodes used for the component to be completed integration at the same time. As the other approach, if each component is fully optimized and can be implemented with high parallelization on whole nodes of the Earth Simulator, sequential coupling shows much higher computational

efficiency. In this coupling scheme, the inconsistent through coupling would be avoided. After integration with the atmosphere component, averaged fluxes during the integration have to be stored in memory until finishing of the following integration with the ocean component. Since MSSG-A and MSSG-O are assumed to be fully optimized and parallelized in this study, the serial coupling scheme is adopted as shown in Fig. 6.

# 4 Computational Performance on the Earth Simulator

## 4.1 Cost Balance and Communication Cost of MSSG-A

Cost of each component of MSSG and communication between atmosphere and ocean components are shown in Table1. Horizontal resolution of the atmospheric component is same as it of the oceanic component, because total amount of fluxes throughout communications between the atmosphere and ocean components should be conserved during whole integration. The shared memory architecture within each node is used for communication of fluxes between atmosphere and ocean components. Therefore, Table 1 shows that low cost of communication due to the shared memory has been achieved. Furthermore, Table 1 suggests that optimization of an atmospheric component plays an important role in order to realize high performance computation in MSSG.

Table 2 shows cost balance of each processes in MSSG-A for global simulations. Cost of dynamical core processes is dominant in MSSG-A. As the results of optimization according to the optimization described in 3.3, computational performance statistics of dynamical core on the Earth Simulator, 51.5Tflops had attained and was estimated by about 39.2% of the theoretical peak of the Earth Simulator. Cost of communications in physical processes is relatively high. It suggests there might be cost imbalance among nodes. The cost imbalance reveled in cloud microphysics processes, therefore, we examined introducing a balancer scheme, which can change decomposition region to be divided to each node, to relax cost imbalance among processes. However, cost imbalance has not been improved. It is still an open problem to improve cost performance in physical processes.

## 4.2 Efficiency of Overlapped Computation in MSSG-O

In MSSG-O, overlapped computation is introduced as mentioned in section 3.5. The efficiency is shown in Table 3. After introducing it, total elapsed time decreases with

**Table 1.** Cost balance of the Earth Simulator in MSSG

| Component | Elapsed time (sec) | Ratio to total elapsed time |
|---|---|---|
| **Atmosphere component; MSSG-A** | **460.641** | **97.09%** |
| **Ocean component; MSSG-O** | **13.548** | **2.86%** |
| **Data exchanging for coupling** | **0.256** | **0.05%** |
| **Total elapsed time of MSSG** | **474.445** | **-** |

**Table 2.** Cost balance of the Earth Simulator in MSSG

| Processes / Schemes | Elapsed time (sec) | Ratio to total elapsed time |
|---|---|---|
| Whole processes of MSSG-A | 1773.36 | - |
| Dynamical core processes | 1435.39 | 80.94% |
| Physics processes | 337.97 | 19.06% |
| **Processes / Schemes** | **Ratio to total elapsed time of physics processes** | **Ratio to total elapsed time of physics processes** |
| Cloud micro physics | 67.20% | 12.81% |
| Surface fluxes schemes | 1.80% | 0.34% |
| Subfield variables | 8.35% | 1.59% |
| Radiation schemes | 3.02% | 0.58% |
| Land processes | 1.71% | 0.33% |
| Others | 1.08% | 0.21% |
| Communications | 16.85% | 3.21% |

**Table 3.** Cost efficiency with overlapped computation on the Earth Simulator. T is temperature; S is salinity; and ρ shows density in the ocean.

| Schemes | Overlapped computation | | Computation without overlap | | | |
|---|---|---|---|---|---|---|
| | Total elapsed time (sec) | Ratio to total elapsed time | Total elapsed time (sec) | | Ratio to total elapsed time | |
| Whole ocean component | 576.281 | - | 688.251 | | - | |
| Two dimensional Poison solver | 246.641 | 42.80% | 350.957 | 247.323 | 50.99% | 35.94% |
| Computation of T, S and ρ | | | | 103.634 | | 15.06% |

about 20%. The parallelization is implemented by using microtasking architecture within each node, communication cost of data transfer among each task can be neglect by using shared memory in a node.

## 5 Simulation Results

### 5.1 Results from MSSG-A

Global simulation has been performed to validate physical performance under the condition of 1.9 km horizontal resolution and 32 vertical layers. 72 hours integration was executed with the atmospheric component. Initialized data was interpolated at 00UTC08Aug2003 from Grid Point Value (GPV) data provided by Japan Meteorological Business Support Center. Sea surface data was also made by GPV data at 00UTC08Aug2003 and fixed during the simulation. Precipitation distribution for global has been presented in Fig.7. Fig.7 shows averaged precipitation one hour

before 00UTC10Aug2003. The unit is mm per hour. Precipitation distribution has been brought by working of cloud microphysics and is comparable to observation data. In this simulation, diurnal cycle of precipitation in Indonesian region and fine structure of fronts are captured.

Regional coupled simulations for physical validation has been performed with one way nesting from 11 km global to Japanese region with 2.7km horizontal resolution. Horizontal resolution was set the same condition in oceanic component of MSSG. Initialized data was interpolated by using GPV data at 00UTC08Aug2003 provided by Japan Meteorological Business Support Center Boundary condition was made by interpolation the above simulation with 5.5km horizontal resolution. Sea surface temperature was also fixed to data at 00UTC08Aug2003 during the simulation. 72-hours integration has been performed. Fig.8 shows the result after 72 hours integration. White gradation distribution in the typhoon shows cloud water distribution corresponding to cloud distribution. Fine rain band structure has been captured shown in Fig. 8. In the ocean, distribution of sea surface temperature (SST) is presented as well in Fig.8. SST responses to a strong wind due to typhoon and disturbance of SST are simulated. Not only SST but also vertical velocity in the ocean has been dramatically changed in Kuroshio region (data not shown).

### 5.2 Results from MSSG-O

In standalone oceanic components of MSSG, as validation simulation, 15-years integration with 11km horizontal resolution and 40 vertical layers has been executed for the North Pacific basin and region between the equator and 30ºS in CASE4 Surface heat fluxes and boundary data are computed from climatological data provided by World Ocean Atlas (WOA). Momentum fluxes are obtained by interpolating from climatological data by NCAR. Fig. 7 shows temperature distribution at 15 m depth from the surface, which is corresponding to the second layer from the surface.

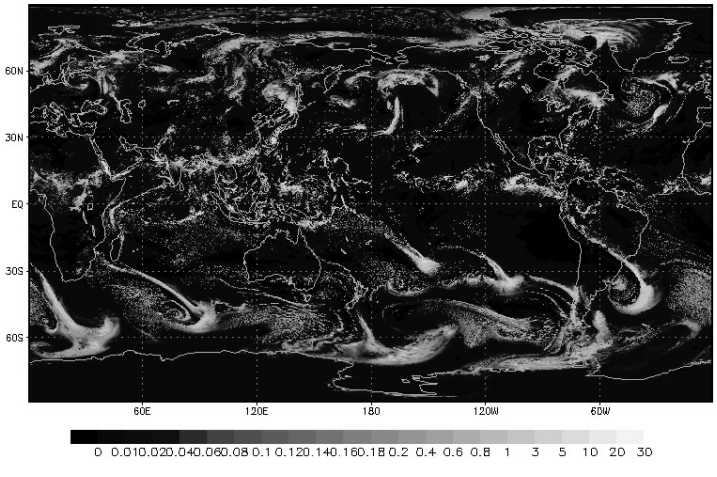



**Fig. 7.** Global precipitation distribution with the atmospheric component (mm/h)

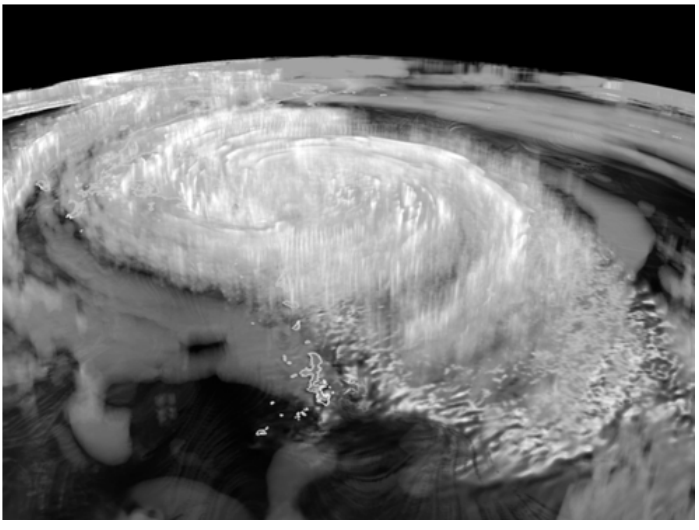


**Fig. 8.** Regional validation results with the atmospheric component. Colored distribution shows the precipitation (mm/hour).

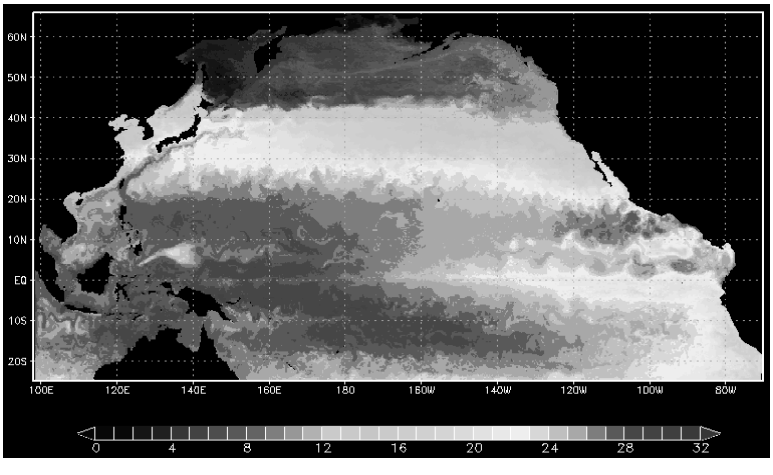



**Fig. 9.** Snap shot results from regional simulations with the MSSG-O after 15 years integration. SST (℃) at 15m depth from surface are distributed.

### 5.3 Urban Scale Simulations

We especially focus on developing models of urban scale phenomena which will be one of the key parts of *seamless* simulation. The impacts of Large Eddy Simulation scheme for boundary layer in MSSG was explored by simulation. Marunouchi area in the center of Tokyo was selected for the simulation. Experiments were performed with both 5m horizontal and vertical resolution in MSSG-A. The initial state was settled at 15:00 on 5$^{th}$ August in 2005. Initial thermal condition was set taking account


**Fig. 10.** Temperature distribution of urban scale simulations for Marunouchi are in Tokyo

of shade in a day. In simulations with 5m horizontal resolution, buildings can be resolved with anthropogenic heating source. Fig. 10 shows a snap shot of horizontal temperature distribution during nonstationary computation from the initial time 15:00. Dynamics of thermal plume have been well represented.

## 6 Conclusions and Perspectives

The developments of MSSG was successfully almost completed with the high computational performance, although further optimization remains in communications. Simulation results with each time/space scale encourage us to start *seamless* simulations with MSSG. As challenging issues, more various forecasting experiments is going to be performed and more longer integration will be executed in order to estimate the accuracy of forecasting.

## References

1. Satomura, T., Akiba, S.: Development of high- precision nonhydrostatic atmospheric model (1): Governing equations. Annuals of Disas. Prev. Res. Inst., 331–336 (2003)
2. Lilly, D.K.: On the numerical simulation of buoyant convection. Tellus 14, 148–172 (1962)
3. Smagorinsky, J., Manabe, S., Holloway Jr., J.L.: Numerical results from a nine level general circulation model of the atmosphere. Monthly Weather Review 93, 727–768 (1965)
4. Zhang, D., Anthes, R.A.: A High-Resolution Model of the Planetary Boundary Layer - Sensitivity Tests and Comparisons with SESAME-79 Data. Journal of Applied Meteorology 21, 1594–1609 (1982)

5. Blackadar, A.K.: High resolution models of the planetary boundary layer. In: Pfafflin, Ziegler (eds.) Advances in Environmental Science and Engineering, vol. 1, pp. 50–85. Gordon and Breach Publ. Group, Newark (1979)
6. Reisner, J., Ramussen, R.J., Bruintjes, R.T.: Explicit forecasting of supercooled liquid water in winter storms using the MM5 mesoscale model. Quart. J. Roy. Meteor. Soc (1998)
7. Kain, J.S., Fritsch, J.M.: Convective parameterization for mesoscale models: The Kain-Fritsch Scheme. The Representation of Cumulus Convection in Numerical Models of the Atmosphre, Meteor. Monogr. 46, 165–170 (1993)
8. Fritsch, J.M., Chappell, C.F.: Numerical prediction of convectively driven mesoscale pressure systems, Part I: Convective parameterization. J. Atmos. Sci. 37, 1722–1733 (1980)
9. Davies, H.C.: A lateral boundary formulation for multi-level prediction models. Quart. J. R. Met. Soc. 102, 405–418 (1976)
10. Marshall, J., Hill, C., Perelman, L., Adcroft, A.: Hydrostatic, quasi-hydrostatic, and nonhydrostatic ocean modeling. Journal of Geophysical Research 102, 5733–5752 (1997)
11. Marshall, J., Adcroft, A., Hill, C., Perelman, L., Heisey, C.: A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers. Journal of Geophysical Research 102, 5753–5766 (1997)
12. Gill, A.: Atmosphere-Ocean dynamics. Academic Press Inc., London (1982)
13. Mellor, G.L., Yamada, T.: A hierarchy of turbulence closure models for planetary boundary layers. Journal of Atmospheric Sciences 31, 1791–1806 (1974)
14. Kageyama, A., Sato, T.: The "Yin-Yang Grid": An Overset Grid in Spherical Geometry. Geochem. Geophys. Geosyst. 5, 9005 (2004), doi:10.1029/2004GC000734
15. Gal-Chen, T., Somerville, R.C.J.: On the use of a coordinate transformation for the solution of the Navier-Stokes equations. Journal of Computational Physics 17, 209–228 (1975)
16. Peng, X., Xiao, F., Takahashi, K., Yabe, T.: CIP transport in meteorological models. JSME international Journal (Series B) 47(4), 725–734 (2004)
17. Wicker, L.J., Skamarock, W.C.: Time-splitting methods for elastic models using forward time schemes. Monthly Weather Review 130, 2088–2097 (2002)
18. Komine, K.: Validation Results from Non-hydrostatic Atmospheric Simulations. In: 2005 SIAM Conference on Computational Science and Engineering, Florida, USA (Febuary 2005)
19. Ohdaira, M., Takahashi, K., Watanabe, K.: Validation for the Solution of Shallow Water Equations in Spherical Geometry with Overset Grid System. In: Spherical Geometry. The 2004 Workshop on the Solution of Partial Differential Equations on the Sphere, Yokohama, p. 71 (July 2004)
20. Takahashi, K., et al.: Proc. 7th International Conference on High Performance Computing and Grid in Asia Pacific Region, p. 487 (2004)
21. Takahashi, K. et al.: Non-Hydrostatic Atmospheric GCM Development and its computational performance,
`http://www.ecmwf.int/newsevents/meetings/workshops/2004/high_performance_computing-11th/presentations.html`
22. Stuben, K.: A Review of Algebraic Multigrid, GMD Report 96 (1999)

# Author Index


Ablayev, Farid 1
Adamidis, Panagiotis 390
Alfonsi, Giancarlo 404

Baba, Yuya 484
Bandini, Stefania 125
Bandman, Olga 140
Baranov, Anton 459
Belyaev, Alexey B. 98
Bespalov, Dmitry 473
Bessonov, Oleg 13
Bobchenkov, Alexander 368
Borisenko, Andrey 417
Bratosin, Carmen 224
Brzeziński, Jerzy 238

Cao, Jianwen 257
Cazzaniga, Paolo 62
Chojnacki, Bartosz 431
Ciliberti, Stefania A. 404
Clarke, David 332

Danilecki, Arkadiusz 244
Degano, Pierpaolo 23
Dementyeva, Ekaterina 444
Désérable, Dominique 152
Ding, Zuohua 257
Dordopulo, Alexey 272
Dwornikowski, Dariusz 238
Dziurzanski, Piotr 431

Elokhin, Vladimir 204
Evtyushkin, Dmitry 284

Fast, Irina 390
Ferrari, Gian-Luigi 23
Fuchigami, Hiromitsu 484
Fujita, Hamido 290

Gorlatch, Sergei 417
Goto, Koji 484

Hoffmann, Rolf 152
Hołenko, Mateusz 244

Ivashko, Evgeny 437

Javed, Noman 40

Kalgin, Konstantin 166
Kalyaev, Igor 272
Karepova, Evgeniya 444
Kegel, Philipp 417
Kida, Shinichiro 484
Kireev, Sergey 290
Kobusińska, Anna 244
Kobusiński, Jacek 238

Lastovetsky, Alexey 332
Lazarev, Alexander 459
Levin, Ilya 272
Loulergue, Frédéric 40
Ludwig, Thomas 390

Maj, Carlo 62
Maka, Tomasz 431
Malyshkin, Victor E. 53, 290
Mancini, Marco 404
Marchenko, Mikhail 302
Marowka, Ami 317
Mauri, Giancarlo 62
Medvedev, Yuri 175
Merelli, Ivan 62
Mezzetti, Gianluca 23
Milanesi, Luciano 62
Milde, Benjamin 452
Mosca, Ettore 62
Moskovsky, Alexander 384
Mostéfaoui, Achour 74
Müller, Thomas 88

Nepomniaschaya, Anna 182
Nikitina, Natalia 437

Onishi, Ryo 484
Ostapkevich, Mike 192

Panfilov, Peter 284
Perepelkin, Vladislav A. 53
Pescini, Dario 62

Piskunov, Sergey 192
Ponomarev, Dmitry 284
Posypkin, Mikhail 473
Primavera, Leonardo 404

Raynal, Michel 74
Rubagotti, Federico 125
Rusin, Evgeny V. 467
Rychkov, Vladimir 332

Salnikov, Anton 459
Schneider, Michael 452
Semenov, Alexander 473
Seredynski, Franciszek 347
Sharifulina, Anastasia 204
Shen, Hui 257
Shimura, Kenichiro 125
Shoshmina, Irina V. 98
Sidorova, Natalia 224
Slasten, Liubov 272
Smajic, Jasmin 88
Stasenko, Alexander 110
Sugimura, Takeshi 484
Switalski, Piotr 347
Szychowiak, Michał 244

Takahashi, Keiko 484
Tarkov, Mikhail S. 358
Toporkov, Victor 368
Toporkova, Anna 368
Trinitis, Carsten 88
Tselishchev, Alexey 368
Tyutlyaeva, Ekaterina 384

Umeo, Hiroshi 210

van der Aalst, Wil 224
Vasiliev, Alexander 1
Vizzari, Giuseppe 125

Yanagihara, Takashi 210
Yemelyanov, Dmitry 368

Zaikin, Oleg 473
Zierhoffer, Piotr 244