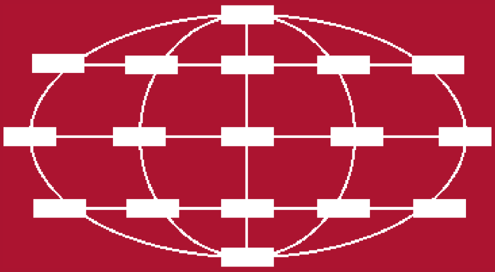
Victor Malyshkin (Ed.)

LNCS 7979

Parallel Computing Technologies

12th International Conference, PaCT 2013
St. Petersburg, Russia, September/October 2013
Proceedings



Springer

Victor Malyshkin (Ed.)

Parallel Computing Technologies

12th International Conference, PaCT 2013
St. Petersburg, Russia
September 30 - October 4, 2013
Proceedings

Springer

Volume Editor

Victor Malyshkin
Russian Academy of Science
Institute of Computational Mathematics and
Mathematical Geophysics
Supercomputer Department
630090 Novosibirsk, Russia
E-mail: malysh@ssd.sscc.ru

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39957-2 e-ISBN 978-3-642-39958-9
DOI 10.1007/978-3-642-39958-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013943816

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-2013 (Parallel Computing Technologies) conference was a four-day event held in St. Petersburg. This was the 12th international conference in the PaCT series. The conferences are held in Russia every odd year. The first conference, PaCT-91, 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; in Kazan, September 19–23. Since 1995, all the PaCT proceedings have been published by Springer in the LNCS series. PaCT-2013 was jointly organized by the Institute of Computational Mathematics and Mathematical Geophysics of Russian Academy of Sciences and the Saint Petersburg State Polytechnical University (SPbSPU). The aim of PaCT-2013 was to give an overview of new developments, applications, and trends in parallel computing technologies. We sincerely hope that the conference helped our community to deepen its understanding of parallel computing technologies by providing a forum for an exchange of views between scientists and specialists from all over the world. The conference attracted 88 participants from around the world. Authors from 18 countries submitted 83 papers. Of these, 41 were selected for the conference as regular papers; there were also two invited speakers. All the papers were reviewed by at least three international referees. Many thanks to our sponsors: Russian Academy of Sciences, Russian Fund for Basic Research, Closed Joint Stock Company Intel A/O.

October 2013 Victor Malyshkin

Organization

PaCT 2013 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 Saint Petersburg State Polytechnical University (SPbSPU).

Organizing Committee

Conference Co-chairs

Dmitry Raychuk - SPbSPU (St. Petersburg)
Nikolay Shabrov - SPbSPU (St. Petersburg)
Victor Malyshkin - ICM&MG (Novosibirsk)

Organizing Committee Members

Dmitry Arseniev, SPbSPU	Vladimir Korneev, ICM&MG
Svetlana Achasova, ICM&MG	Mikhail Ostapkevich, ICM&MG
Olga Bandman, ICM&MG	Vladislav Perepelkin, ICM&MG
Yuri Boldyrev, SPbSPU	Valentina Markova, ICM&MG
Maxim Gorodnichev, ICM&MG	Yuri Medvedev, ICM&MG
Larisa Ivanyshina, SPbSPU	Vladimir Sinepol, SPbSPU
Yury Karpov, SPbSPU	Evgeny E.M. Smirnov, SPbSPU
Sergey Kireev, ICM&MG	Mikhail Strelets, SPbSPU
Anastasia Kireeva, ICM&MG	Georgy Schukin, ICM&MG

Program Committee

Victor Malyshkin	Russian Academy of Sciences, Chair
Farid Ablaev	Kazan Federal University, Russia
Sergey Abramov	Russian Academy of Sciences, Russia
Farhad Arbab	Leiden Institute for Advanced Computer Science, The Netherlands
Stefania Bandini	University of Milano-Bicocca, Italy
Olga Bandman	Russian Academy of Sciences
Thomas Casavant	University of Iowa, USA
Pierpaolo Degano	University of Pisa, Italy
Dominique Désérable	National Institute for Applied Sciences, Rennes, France
Sergei Gorlatch	University of Münster, Germany
Yuri G. Karpov	St.Petersburg State Polytechnical University, Russia

Alexey Lastovetsky	University College Dublin, Ireland
Thomas Ludwig	University of Hamburg, Germany
Mikhail Marchenko	Russian Academy of Sciences
Giancarlo Mauri	University of Milano-Bicocca, Italy
Nikolay Mirenkov	University of Aizu, Japan
Dana Petcu	West University of Timisoara, Romania
Viktor Prasanna	University of Southern California, USA
Michel Raynal	Research Institute in Computer Science and Random Systems, Rennes, France
Bernard Roux	National Center for Scientific Research, France
Mitsuhisa Sato	University of Tsukuba, Japan
Carsten Trinitis	University of Bedfordshire, UK
Mateo Valero	Technical University of Catalonia, Spain
Roman Wyrzykowski	Czestochowa University of Technology, Poland
Laurence T. Yang	St. Francis Xavier University, Canada

Additional Reviewers

Sebastian, Albers
Farhad, Arbab
Stefania, Bandini
Olga, Bandman
Oleg, Bessonov
Thomas, Casavant
Hsuan-Yi, Chu
Marco, Danelutto
Marco, Danelutto
Pierpaolo, Degano
Pierpaolo, Degano
Dominique, Désérable
Baiardi, Fabrizio
Victor, Gergel
Sergei, Gorlatch
Maxim, Gorodnichev
Konstantin, Kalgin
Yuri, Karpov
Sergey, Kireev
Alok, Kumbhare
Alexey, Lastovetsky
Minh, Lê
Victor, Malyshkin
Mikhail, Marchenko
Giancarlo, Mauri
Dominik, Meiländer
Alessio, Micheli
Alessio, Micheli
Mikhail, Ostapkevich
Vladislav, Perepelkin
Dana, Petcu
Viktor, Prasanna
Michel, Raynal
Nikolay, Shilov
Michel, Steuwer
Massimo, Torquati
Massimo, Torquati
Carsten, Trinitis
Mateo, Valero
Roman, Wyrzykowski
Laurence T., Yang

Sponsoring Institutions

Russian Academy of Sciences,
Russian Fund for Basic Research,
Closed Joint Stock Company Intel A/O

Bi-objective Optimization for Scheduling in Parallel Computing Systems

Anthony A. Maciejewski

Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, Colorado, USA
aam@ColoState.edu
www.engr.colostate.edu/~aam

Abstract. Most challenging engineering problems consider domains where there exist multiple objectives. Often the different objectives will conflict with each other and these conflicts make it difficult to determine performance trade-offs between one objective and another. Pareto optimality is a useful tool to analyze these trade-offs between the two objectives. To demonstrate this, we explore how Pareto optimality can be used to analyze the trade-offs between makespan and energy consumption in scheduling problems for heterogeneous parallel computing systems. We have adapted a multi-objective genetic algorithm from the literature for use within the scheduling domain to find Pareto optimal solutions. These solutions reinforce that consuming more energy results in a lower makespan, while consuming less energy results in a higher makespan. More interestingly, by examining specific solution points within the Pareto optimal set, we are able to perform a deeper analysis about the scheduling decisions of the system. These insights in balancing makespan and energy allow system administrators to efficiently operate their parallel systems based on the needs of their environment. The bi-objective optimization approach presented can be used with a variety of performance metrics including operating cost, reliability, throughput, energy consumption, and makespan. More information about this research can be found at http://www.engr.colostate.edu/~aam.[1]

[1] This research was supported by the National Science Foundation under grant number CNS-0905399.

Robust Computing Systems

H.J. Siegel

Department of Electrical and Computer Engineering
and Department of Computer Science
Colorado State University
Fort Collins, Colorado, USA
`HJ@ColoState.edu`
`www.engr.colostate.edu/~hj`

Abstract. What does it mean for a system to be robust? How can robustness be described? How does one determine if a claim of robustness is true? How can one decide which of two systems is more robust? We explore these general questions in the context of heterogeneous parallel and distributed computing systems. A critical research problem is how to allocate heterogeneous resources to tasks to optimize a given performance measure. It is important for system performance to be robust against uncertainty. To accomplish this, we present a stochastic model for deriving the robustness of a resource allocation. The robustness of a resource allocation is quantified as the probability that a user-specified level of system performance can be met. We show how to use historical data to build a probabilistic model to evaluate the robustness of resource assignments and how to design resource management techniques that produce robust allocations. The robustness analysis approach presented can be applied to a variety of computing and communication system environments, including parallel, distributed, cluster, grid, Internet, cloud, embedded, multicore, content distribution networks, wireless networks, and sensor networks. Furthermore, the robustness model is generally applicable to design problems throughout various scientific and engineering fields. More information about this research can be found at www.engr.colostate.edu/~hj.[1]

[1] This research was supported by the National Science Foundation under grant number CNS-0905399 and the Colorado State University George T. Abell Endowment.

Table of Contents

Parallel Programming Tools

Strassen's Communication-Avoiding Parallel Matrix Multiplication Algorithm for All-Port 2D Torus Networks

Cesur Baransel

Saltus Yazılım Ltd., Hacettepe University Technopolis, Ankara, Turkey
cesur@ada.net.tr

Abstract. A parallel implementation of Strassen's matrix multiplication algorithm is proposed for massively parallel supercomputers with 2D all-port torus interconnection networks. The proposed algorithm employs conflict-free routing patterns and operates on completely-connected subnetworks in order to reduce the latency cost L of the algorithm down to $L(n,P) = 6\log_7 P$ on all-port 2D torus architectures having P nodes, which compares favorably to the latency cost $L(n,P) = 36\log_7 P$ of the recently proposed CAPS algorithm.

Keywords: Communication-Avoiding, 2.5D Matrix Multiplication Algorithm, Fast Matrix Multiplication, Strassen's Matrix Multiplication, Parallel Processing, Torus Interconnection Networks, 2D Torus.

1 Introduction

In sequential matrix multiplication of two square matrices of order $n \times n$, computing a single element of the product matrix requires n multiplications followed by $(n-1)$ additions. Thus, $(2n^3 - n^2)$ scalar operations in total should be performed to complete the multiplication. In 1969, Strassen proposed a new formulation of matrix multiplication which reduces the number of scalar operations from $(2n^3 - n^2)$ down to $\left(7n^{\log_2 7} - 6n^2\right)$ where n is a power of two [12]. Strassen's formulation makes it possible to multiply two 2×2 matrices with 7 multiplications instead of 8 and can be recursively applied to the multiplication of submatrices until 2×2 submatrices are reached. When submatrices are distributed among the processors, intermediate results need to be communicated across the network. In recent studies, it has been argued that given the status of the current technology, communication costs are and will continue to be the limiting factor in the design of efficient parallel algorithms and so-called "Communication-Avoiding Algorithms" are proposed as a solution [4]. The basic idea behind the communication-avoiding algorithms is to minimize communications between processors by replicating data beyond the required minimum and also reformulating the communication patterns specified within the algorithm. In an earlier publication, we have proposed an implementation of Strassen's algorithm specifically designed for wormhole-routed, 2D, all-port, torus interconnection networks [3]. In this paper, we present a novel communication-avoiding matrix multiplication algorithm for the same architecture based on a variant of Strassen's formulation. Unlike some recent proposals on the same subject, we address the topological aspects of the algorithm in detail and provide specific conflict-free routing patterns which are instrumental in decreasing the overall number of messages.

This paper is structured as follows. A short review of the previous work is provided in the next section. Section 3 gives the details of the proposed algorithm and Section 4 contains the performance analysis. The paper ends with conclusions.

2 Problem Statement and Previous Work

Cannon's parallel matrix multiplication algorithm can effectively parallelize the task of performing the involved $(2n^3 - n^2)$ scalar operations over $p = \sqrt{p} \times \sqrt{p}$ processors with the following computational and communication costs [5]:

$$T_{Cannon} = 2\sqrt{p}\ (\alpha + \beta \frac{n^2}{p}) + \gamma \frac{n^3}{p} \tag{1}$$

where α is the latency expressed in seconds, β is the inverse of the bandwidth expressed in words-per-second and γ is the time required to perform a scalar operation expressed in seconds. The above formula includes the cost of the initial and final alignments each of which can be completed in at most $\sqrt{p}/2$ steps on 2D torus network. Omitting alignment phase, the overall cost becomes:

$$T_{Cannon} = \sqrt{p}\ (\alpha + \beta \frac{n^2}{p}) + \gamma \frac{n^3}{p} \tag{2}$$

When communication overlaps computation, the above formula can be interpreted in two ways. If the network is sufficiently fast, processors can find the next operands already delivered when they complete the local scalar operations. In that case, the lower bound of the algorithm can be stated as $\Omega\left(n^3/p\right)$. On the other hand, if processors are faster, they have to wait for the delivery of the next operands after completing the local scalar operations, and the communication cost determines the lower bound of the algorithm. Therefore, we can write:

$$T_{Cannon} = \max\left(\left[\sqrt{p}\ (\alpha + \beta \frac{n^2}{p})\right], \left[\gamma \frac{n^3}{p}\right]\right) \tag{3}$$

It is known that, $\gamma \ll \beta \ll \alpha$ for the current multiprocessor interconnection networks[1] and especially the network latency is not likely to improve significantly in the near future [2]. Thus, in view of the widening gap between network latency and processor speeds, it is getting increasingly more important to reduce the number of messages and the message lengths involved in an application.

Recently, some communication avoiding matrix multiplication algorithms have been proposed in the literature [2], [4], [8] and [11]. Of particular interest here is the CAPS (Communication-Avoiding Parallel Strassen) Algorithm which reportedly performs asymptotically better than any previous classical or Strassen-based parallel algorithm [2], [8]. In these works, authors give an algorithm which is communication-optimal according to the lower bounds on communication costs derived in [1], [6] and [11]. However, they also indicate that "on some architectures it may be more important to consider the topology of the network and redesign the algorithm to minimize contention" which they have not done. In the next section, we will address these issues and present a solution which operates over conflict-free communication paths and minimizes message counts at the expense of some extra memory usage.

3 The Proposed Algorithm

In this section, we give the details of the proposed algorithm in three different settings. In the first case, 7 processors are mapped to a 3×3 torus and we show that Strassen's algorithm can be completed in 2 communication steps provided that no further recursions are involved. In the second case, 49 processors are mapped to a 7×7 torus where we perform one recursion and the total number of communication steps is 6. In the third case, 4 groups of 7 processors are mapped to a 6×6 torus and the total number of communication steps is 7 or 15 depending on the amount of memory used. In the last case, we also perform one recursion but divide it into two sequential steps. These three cases can be used in different combinations to fit the Strassen's algorithm into a given machine rather than requiring a machine with 7^kx7^k processors for executing (k–1) recursive calls to the algorithm.

3.1 The Basic 2×2 Multiplication Algorithm

The proposed implementation of Strassen's matrix multiplication algorithm for multiplying two 2×2 matrices uses a 7-processors tile mapped to a 3×3 torus. We call it the *base case of Strassen*. Figure 1 shows how summation and multiplication tasks are assigned to the processors.

[1] For a machine with an effective floating-point rate of 10 Gflop/s per processor, network latency of 10 μs, and network bandwidth of 4 GB/s, these parameters can be calculated as $\gamma = 10^{-10} s$, $\alpha = 10^{-5} s$ and $\beta = 2.10^{-9} s / word$ where a word consists of 4 bytes.

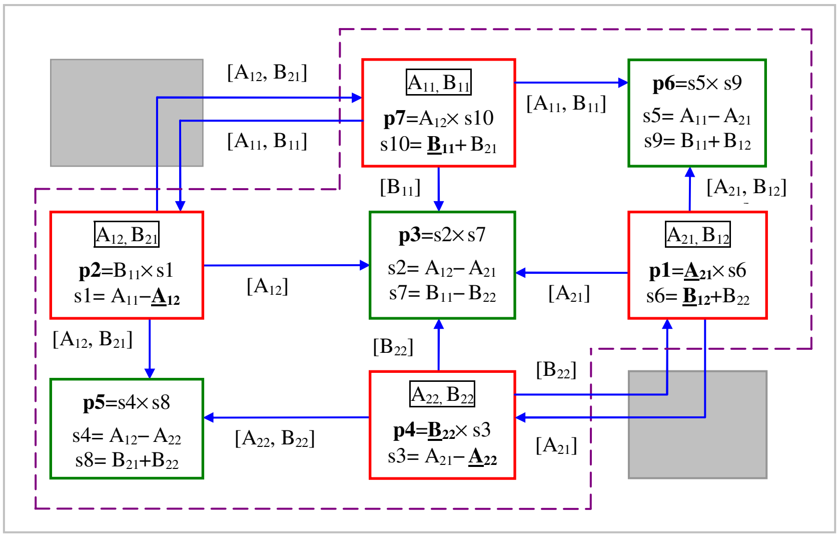


Fig. 1. Strassen's matrix multiplication steps for 2×2 matrices. Underlined elements are already in place.

The initial locations of the elements of the matrices to be multiplied, namely A11, A12, A21, A22, B11, B12, B21 and B22, are shown in Figure 1 (red nodes in color figure). Figure 1 also presents the details of the formulation of the Strassen's method as proposed by Kolen and Bruce [7]. This variant is proved to be more suitable for our approach since it removes the imbalance regarding the usage of data held in processors (as all are now involved in exactly five different sums) and also distributes their indirect involvements in the products of sums more evenly across the processors. Our reasons for choosing this variant are explained in detail in our previous work [3]. Remember that in an all-port torus interconnection network, a node can send and receive messages at four ports in parallel. Figure 1 is supposed to convey the following information visually regarding the algorithm:

1. Assuming the initial alignment in the figure, only one communication step is necessary prior to the calculation of 10 sums and 7 products.
2. Products are calculated in place, without moving sums around. In other words, no communication is required prior to calculating products once the calculations of the sums are finished.
3. For calculating the elements of the products matrix C, one additional communication step is required (Figure 2).
4. The execution of the Strassen's algorithm without any further recursions is completed in two communications and two calculation steps.
5. Nodes store at most four matrix elements, instead of the minimum two. This is the extent of the extra memory usage involved.
6. Communication and computation steps cannot be overlapped.

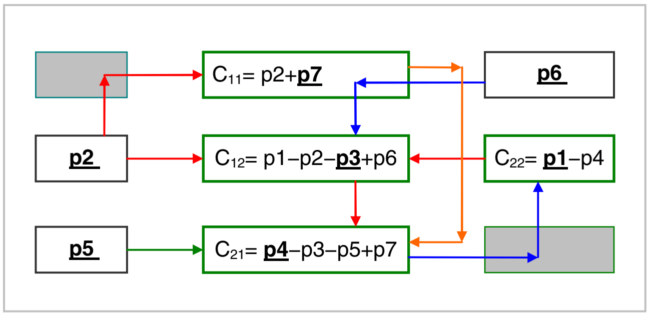


Fig. 2. Calculating the elements of the products matrix C from 7 products. Underlined elements are already in place.

The latency (L) and bandwidth (B) cost of the base case can be computed as follows. Assuming $N = n \times n$ input matrices, one processor stores N/4 elements of each input matrix. The elements of product matrices are also composed of N/4 elements. There are 2 communication steps in total. At the first step, some messages can be $2 \times N/4$ elements long. In the second step, transmitted messages are always composed of N/4 elements. Thus for a node, we can write:

$$L(N) = 2\alpha, \quad B(N) = 3\left(\frac{N}{4}\right)\beta = 0.75N\beta \tag{4}$$

3.2 Performing Recursion

Figure 3 shows how the first level of recursion is applied where 49 processors mapped onto 7×7 torus. Recursion starts in the central blue tile and the input matrices for the seven products are transmitted to the centers of the seven tiles which are indicated as c1 to c7 in the figure. Then, with another transmission step the initial alignment is achieved within each tile. Thus, going into a recursion level requires two transmission steps. Since results will follow the same path in reverse direction, another two transmission steps are necessary to return from the recursion. In other words, each recursion requires 6 communication and 2 computation steps and none of these steps can overlap each other.

The latency (L) and bandwidth (B) costs of single recursion can be computed as follows. Assuming $N = n \times n$ input matrices at the beginning of the recursion, each of the 7 product submatrices will be composed N/4 elements. Once the recursion is executed, one element from each of the two input matrices will be transmitted to center nodes in a single communication step. Since one element contains the N/4 elements of the original matrix, the latency and bandwidth costs of this step are:

$$L(N) = \alpha, \quad B(N) = 2\frac{N}{4}\beta \tag{5}$$

Center nodes perform the initial alignment in a single communication step by transmitting messages which contain $2 \times N/16$ elements each. The latency and bandwidth costs of this step are:

$$L(N) = \alpha, \quad B(N) = 2\frac{N}{16}\beta \tag{6}$$

The products are computed as in the base case and resulting matrices are formed. The latency and bandwidth costs of this step are:

$$L(N) = 2\alpha, \quad B(N) = 3\frac{N}{16}\beta \tag{7}$$

The return from the recursion requires two more communication steps in the reverse direction. However in the reverse direction, messages are only N/16 and N/4 elements long. Combining, we have:

$$L(N) = 6\alpha, \quad B(N) = 18\left(\frac{N}{16}\right)\beta = 1.125N\beta \tag{8}$$

In general, the tiles of processors do not overlap and individual matrix multiplications are executed in parallel on 7^k processors after k levels of recursion. To tile a 2D torus with 7-processor tiles recursively, we basically replace each node in the basic building block with a copy of itself. Note that, while a 7×7 tile fit into the corresponding 2D torus perfectly, for configuration allowing an even number of recursions at maximum (such as a 21×21 processor network which permits up to two recursions to be performed) 2/9 of the processors will be left unutilized. The details of the tiling mechanism and how to handle arbitrary matrix sizes are discussed extensively in [3].

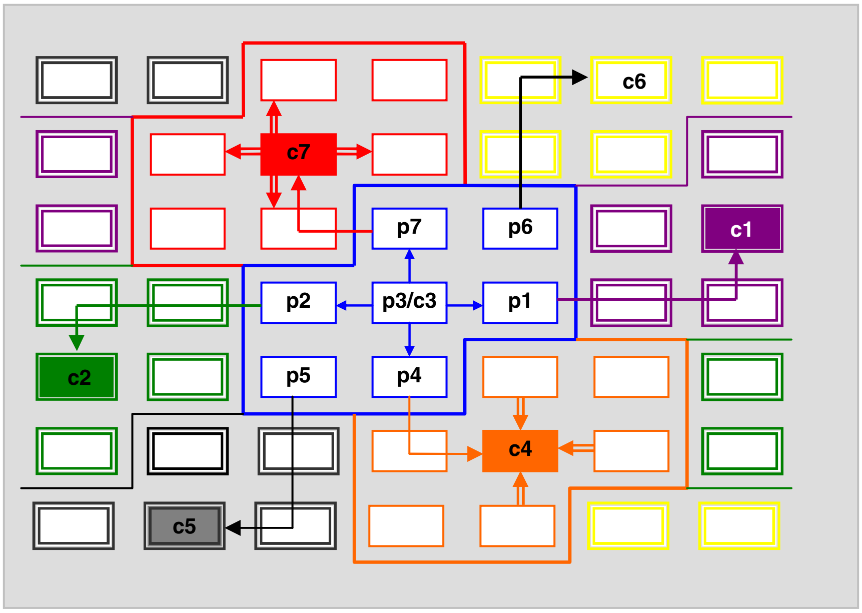
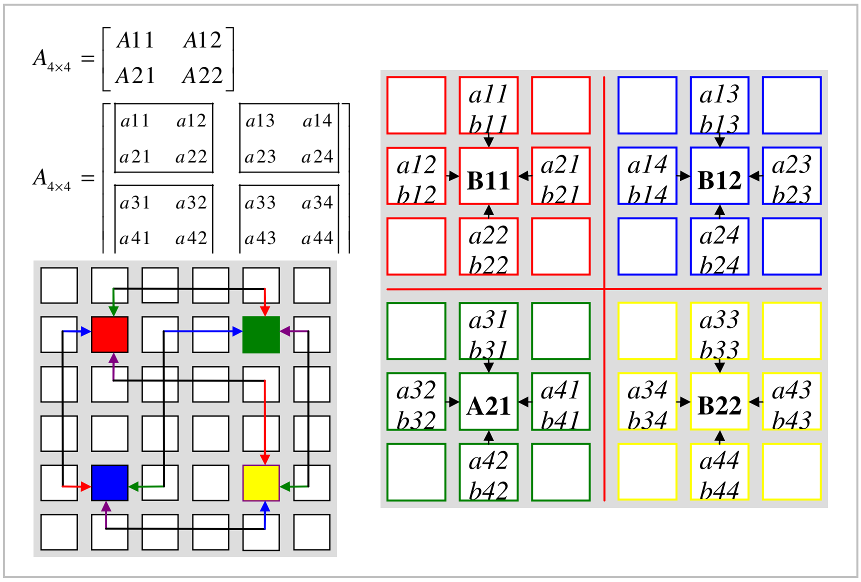


Fig. 3. First recursion applied to 49 processors mapped onto 7×7 torus

3.3 Matrix Multiplication on a 6×6 Torus

Since Strassen's algorithm breaks down a matrix into 4 submatrices and creates 7 submatrices from each to compute 7 products, it naturally maps into 4 blocks of 7 processors as seen in Figure 4.



$A_{4\times4} = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$
$A_{4\times4} = \begin{bmatrix} \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} & \begin{bmatrix} a13 & a14 \\ a23 & a24 \end{bmatrix} \\ \begin{bmatrix} a31 & a32 \\ a41 & a42 \end{bmatrix} & \begin{bmatrix} a33 & a34 \\ a43 & a44 \end{bmatrix} \end{bmatrix}$
a11 b11
a12 b12
B11
a21 b21
a22 b22
a13 b13
a14 b14
B12
a23 b23
a24 b24
a31 b31
a32 b32
A21
a41 b41
a42 b42
a33 b33
a34 b34
B22
a43 b43
a44 b44

Fig. 4. Strassen's Algorithm on 6×6 torus

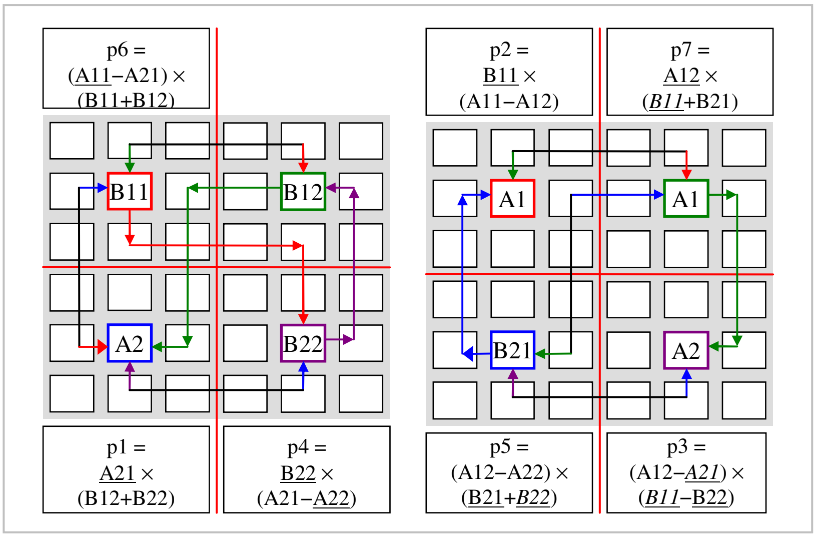


Fig. 5. Strassen's Algorithm on 6×6 torus in two consecutive phases. Underlined elements are already in place.

In this case, the algorithm is completed in two consecutive phases. The steps for the first phase of the algorithm, and associated latency and bandwidth costs are provided in Table 1.

Table 1. Steps of the first phase

Step	Process	Latency, Bandwidth Costs
1	{A21, B11, B12, B22} are formed in the central nodes (Figure 4).	$L(N)=\alpha, \quad B(N)=\frac{N}{16}\beta$
2	{A21, B11, B12, B22} are exchanged among the central nodes in a single communication step (Figure 5). This exchange provides the data for the computation of three products p1, p4 and p6.	$L(N)=\alpha, \quad B(N)=4\left(\frac{N}{4}\right)\beta$
3	Center nodes perform the initial alignment in a single communication step by transmitting messages which contain $2\times N/16$ elements each.	$L(N)=\alpha, \quad B(N)=2\left(\frac{N}{16}\right)\beta$
4	Compute each of the three products in a different block of 7 processors in 2 communication steps (as explained in Section 3.1.).	$L(N)=2\alpha, \quad B(N)=3\left(\frac{N}{16}\right)\beta$
5	Combine product submatrices in the central nodes and move these three products to the nodes where they will be combined to produce the product matrix C.	$L(N)=2\alpha, \quad B(N)=5\left(\frac{N}{16}\right)\beta$

The overall latency and bandwidth costs of the first phase are:

$$L(N)=7\alpha, \quad B(N)=15\left(\frac{N}{16}\right)\beta \approx 0.94N\beta \qquad (9)$$

In the second phase, {A11, A12, A22, B21} are formed and the same process is repeated to compute the products p2, p3, p5 and p7. Following the second phase, one additional communication step is necessary to align the elements of the product matrix C with the elements of the input matrices A and B. The latency and bandwidth costs of the whole process are:

$$L(N)=15\alpha, \quad B(N)=31\left(\frac{N}{16}\right)\beta \approx 1.94N\beta \qquad (10)$$

Alternatively, central nodes in each block create A11, A12, A21, A22, B11, B12, B21 and B22 by gathering data from their immediate neighbors in a single communication step so that the above procedure can be executed only once provided that enough memory is available.

Note that the above algorithm can be also expressed in terms of so-called BFS (breadth-first-step) and DFS (depth-first-step) steps of CAPS. A BFS divides the 7 subproblems among the processors, so that 1/7 of the processors work on each subproblem independently and in parallel. DFS uses all processors on each subproblem solving each one in sequence. Here, we defined two sequential subproblems for calculating 7 products, and let 4 groups of 7 processors work on each subproblem simultaneously and in parallel.

4 Performance Analysis

For dense matrix multiplication, algorithms that minimize both bandwidth and latency costs have already provided in literature. When processors have just enough memory to store only one copy of the input/output matrices across a $\sqrt{p}\times\sqrt{p}$ processor array, Cannon's parallel algorithm[2] is optimal in the sense that it simultaneously achieves perfect load balancing $(\Theta(n^3/p))$, minimizes the bandwidth cost $(\Theta(n^2/\sqrt{p}))$ and minimizes latency $(\Theta(\sqrt{p}))$. If processors are allowed to have additional memory so that local memory size becomes $3n^2/p^{2/3}$ (in effect making $p^{1/3}$ copies of input matrices available across the processor array), so-called *3D algorithms* exist that can effectively balance the load $(\Theta(n^3/p))$, minimize the bandwidth cost $(\Theta(n^2/p^{2/3}))$, and minimize the latency cost $(\Theta(\log p))$. In [11] authors presented a communication-optimal parallel 2.5D matrix multiplication algorithm for configurations having enough memory for storing c copies of the input and output matrices. This algorithm sends $c^{1/2}$ times fewer words and $c^{3/2}$ times fewer messages compared to the 2D (Cannon's) algorithm. The authors label any algorithm which stores c copies of the input matrices across the processor array as "*2.5D*" where $c \in \{1, 2, \ldots \lfloor p^{1/3} \rfloor\}$.

The above bounds are not applicable to Strassen-like fast multiplication algorithms. In classical matrix multiplication algorithm a fixed message length is employed throughout the algorithm. In Strassen's algorithm, the submatrices get smaller at each recursion step and at a certain point the submatrix gets so small that it takes longer to transmit it than to compute the product submatrix directly. At this so-called cutoff point recursion terminates. Consequently, mapping the Strassen's algorithm to processors is more complicated compared to classical algorithms. Recently, a communication-optimal parallel algorithm called CAPS is introduced for Strassen's matrix multiplication [2[, [8].

CAPS uses two different schemes for traversing the recursion tree of Strassen's sequential algorithm. If sufficient memory is available, the UM (Unlimited Memory) scheme divides the 7 subproblems amongst the processors so that each subproblem is computed by 1/7 of the processors independently and in parallel. The LM (Limited Memory) scheme is used to reduce the problem size so that UM scheme can be utilized on each subproblem without exceeding the available memory. UM scheme

[2] Also note that additional memory for two matrix members is necessary for overlapping communication with computation, which increases the total memory requirement at a node up to $5(n^2/p)$ for Cannon's 2D algorithm.

offers lower communication costs than LM scheme by computing the 7 products in parallel on disjoint sets of 7 processors. Since, the redistribution of data is performed by all-to-all communication among the sets of 7 processors, the latency cost LUM is given as $L_{UM}(n,P)=36\log_7 P$ which amounts to $L=72\alpha$ for 49 processors. In Section 3.2, we showed how to perform the same task in 6 steps with the latency cost of $L=6\alpha$ only. The difference stems from the following factors:

1. CAPS model assumes that processors can send/receive just one message to/from one processor at a time whereas a node can send and receive messages at four ports in parallel in an all-port torus interconnection network. This has two important implications. First, four times more messages can be communicated within a single communication step. Second, complete-exchange operation can be performed without resorting to all-to-all broadcasts since all-port torus network usually allows a completely connected subnetwork among 4 processors at a time. Since 7 processors need to exchange information in Strassen's algorithm, a complete exchange operation between them can be completed in only two steps, as demonstrated in Section 3.2.
2. CAPS uses recursive Morton ordering following by a block-cyclic layout in each submatrix to store the input matrices which allows it to store a 28×28 matrix into a 7×7 processor array, each processor having one 4×4 block [2, 8, 9]. We use quadtree partitioning to store the input data into a subset of processors (see Section 3.2 and section 3.3). Although CAPS starts with allocating memory for 14 sums at the very beginning of the algorithm, our approach still requires more memory.
3. CAPS uses Strassen-Winogard formulation which requires 7 linear combinations of each submatrix of A and B prior to the computation of 7 products. However, due to dependencies T5=A12+T3, and S6=S3−B21, the computation of sums takes two computation steps. In our preferred formulation, we can compute 10 sums simultaneously in a single computation step.

Given a $2^k \times 2^k$ square matrix, it is possible to call Strassen's algorithm k times[3] and at the last call all input matrices will be reduced to scalars. However, it is known that executing all recursions levels of Strassen's algorithm until operands become scalars does not yield the best performance. The level that the recursive calls to Strassen's algorithm should be terminated due to communication overhead is called the cutoff point.

Let us reconsider the cases presented in Section 3.2 and Section 3.3. For the first case, assume that we have $2^n \times 2^n$ input matrices initially distributed to $2^k \times 2^k$ grid of processors where $k<n$ and each processor has a $2^{n-k} \times 2^{n-k}$ matrix block. Also assume that this $2^k \times 2^k$ grid is a subset of 7^k processors of a $7^{k/2} \times 7^{k/2}$ torus. Then, we can perform *k* recursions in a row and after the *k*th recursion, submatrix multiplications will be running in parallel on all 7^k processors. Given a cutoff point at the

[3] The first call constitutes the base case and the rest are recursions.

matrix size of $2^{\omega} \times 2^{\omega}$, we have two cases to consider. If $(n-k > \omega)$, we invoke $(l = n-k-\omega)$ of Strassen's recursions and then revert to local computation of submatrices. If $(n-k \leq \omega)$, we only execute the base case of the Strassen's algorithm (i.e, first level of Strassen's algorithm without any recursion) and 7 processors execute local matrix multiplications of $2^{(n-k)/2} \times 2^{(n-k)/2}$ matrices only once.

In the second case, the $2^k \times 2^k$ grid is a subset of a smaller torus having less than 7^k processors. Then we check how many subsequent recursions can be supported by the torus. Suppose that the torus has composed of $P = m \times 7^j$ nodes where $j < k$ and $1 < m < 7$. The so-called hybrid-step approach in CAPS suggests that after taking j BFS steps, the remaining steps should be computed locally in groups of m. However, in section 3.2 where $P = 36 > 5 \times 7^1$, we found out running m groups of j recursions yielded better performance. The result is due to the efficiency of the data exchange via a completely connected subnetwork among $m = 4$ nodes.

Machine parameters are instrumental for determining the cutoff point for Strassen's recursions. For example, given a 7^k processor array and related machine parameters $\gamma = 10^{-10}\text{s}$, $\alpha = 10^{-5}\text{s}$ and $\beta = 2.10^{-9}\ \text{s/word}$, the square matrix size $\omega \times \omega$ at the cutoff point can be determined by solving the following inequality:

$$\omega^3 \gamma \geq 6\alpha + 1.125\,\omega^2 \beta + \gamma \left(\frac{\omega}{2} \right)^3 \text{ which is satisfied for } \omega \geq 98.$$

The above inequality is satisfied when the local classical matrix multiplication takes *longer time* compared to making a recursive call to Strassen's algorithm. It is also possible to use the cost of the local serial Strassen algorithm instead of the serial classical matrix multiplication's cost.

5 Conclusions and Future Work

In this paper, we proposed a parallel algorithm for Strassen's matrix multiplication formulation on 2D, all-port processor arrays. The main conclusion of this paper is to show that it is possible gain significant reduction in the latency cost of the Strassen's algorithm by explicitly controlling the communication paths on all-port 2D torus via ad-hoc routing patterns. This result is especially important for the practical implementations of the algorithm that are supposed to run on machines where the gap between network latency and processor speeds continues to widen.

There are other aspects of the matrix multiplication problem which offers further opportunities for reducing the number of multiplications involved. For example, it is discussed in [10] how to find a proper grouping for a sample matrix product of four matrices A, B, C and D which requires 2680 multiplications when it is grouped as A(B(CD)) and requires 15800 multiplications when it is grouped as ((AB)C)D. Currently, we are working on a similar problem for finding better performing communication patterns on all-port 2D torus networks when the transposes, powers or inverses of input matrices appear more than once in a given matrix multiplication expression. Results will appear in a near-future paper.

References

1. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Brief announcement: Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In: Proc. of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2012, pp. 77–79. ACM (2012)
2. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Communication-optimal parallel algorithm for Strassen's matrix multiplication. In: Proc. of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2012, pp. 193–204. ACM (2012)
3. Baransel, C., İmre, K.: A parallel implementation of Strassen's matrix multiplication algorithm for wormhole-routed all-port 2D torus networks. The Journal of Supercomputing 62(1), 486–509 (2012)
4. Demmel, J., Grigori, L., Hoemmen, M.F., Langou, J.: Communication-optimal Parallel and Sequential QR and LU Factorizations. SIAM J. Sci. Comput. 34(1), A206–A239 (2012)
5. Gramma, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison Wesley (2003)
6. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distrib. Computing 64(9), 1017–1026 (2004)
7. Kolen, J.F., Bruce, P.: Evolutionary Search for Matrix Multiplication Algorithms. In: FLAIRS Conference, pp. 161–165 (2001)
8. Lipshitz, B., Ballard, G., Demmel, J., Schwartz, O.: Communication-avoiding parallel Strassen: Implementation and performance. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, Article 101, 11 p. (2012)
9. Luo, Q., Drake, J.B.: A Scalable Parallel Strassen's Matrix Multiplication Algorithm for Distributed–Memory Computers. In: Proc. of the 1995 ACM Symposium on Applied Computing, pp. 221–226. ACM Press (1995)
10. Michalewicz, Z., Fogel, D.B.: How to Solve It: Modern Heuristics. Springer (2002)
11. Solomonik, E., Demmel, J.: Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 90–109. Springer, Heidelberg (2011)
12. Strassen, V.: Gaussian Elimination is not Optimal. Numerische Mathematik 13, 354–356 (1969)

Timed Resource Driven Automata Nets for Distributed Real-Time Systems Modelling*

Vladimir A. Bashkin[1], Irina A. Lomazova[2,3], and Yulia A. Novikova[2]

[1] Yaroslavl State University, Yaroslavl, 150000, Russia
v_bashkin@mail.ru
[2] National Research University Higher School of Economics,
Moscow, 101000, Russia
i_lomazova@mail.ru, novikova.ulia@gmail.com
[3] Program Systems Institute of the Russian Academy of Science,
Pereslavl-Zalessky, 152020, Russia

Abstract. The paper presents a formalism and a tool for modelling and analysis of distributed real-time systems of mobile agents. For that we use a time extension of our Resource Driven Automata Nets (TRDA-nets) formalism. A TRDA-net is a two-level system. The upper level represents distributed environment locations with a net of active resources. On the lower level agents are modeled by extended finite state machines, asynchronously consuming/producing shared resources through input/output system ports (arcs of the system net). We demonstrate modelling facilities of the formalism and show that specific layers of TRDA-nets can be translated into Timed Automata, as well as into Time Petri nets, thus TRDA-nets integrate merits of both formalisms.

Keywords: Distributed systems modeling, Petri nets, real-time systems, timed automata, verification.

1 Introduction

Modelling and verification of distributed software and hardware system is a great challenge, intensively studied by many researchers. The up-to-date need for modelling distributed systems with dynamic structure, different kinds of agents interaction, mobility of agents and adaptability leads to different extensions of such popular basic formalisms as finite automata and Petri nets. Extending finite automata for modeling concurrent systems is based mainly on communicating automata [1,17,14]. Among a large variety of Petri net extensions we mention 'net-within-net' approach [7,10,12,13,15,20], where tokens may be Petri nets themselves and thus have their own structure and activity. This approach proved to be good for modelling hierarchy, mobility and adaptivity.

Another important perspective is connected with time, when reliability and safety properties depend, to a large extent, on the timing aspects. Among the

* This work is supported by the Basic Research Program of the National Research University Higher School of Economics.

most popular time dependent models are timed automata (TA) [2] and Time Petri nets (TPN) [16].

This paper presents a formalism of Timed Resource Driven Automata nets (TRDA-nets) for modelling time dependent distributed systems with asynchronous interaction of mobile agents. This formalism inherits primality from finite automata, distributed spatiality from Petri nets, resource dependency and dynamic structure from nets of active resources [3,4], and hierarchy/modularity of 'nets-within-nets' approach from nested Petri nets [13,15,11]. Meanwhile TRDA-nets have a relatively simple and natural syntax, and can be effectively translated into widely-excepted formats.

Nets of active resources (AR-nets) were introduced in [3] as a formalism for modeling distributed systems from resource perspective, when active components and resources are not distinguished. AR-nets formalism is in some sense dual to ordinary Petri nets: in AR-nets sets of Petri net transitions and places are united into a single set of *nodes*, but each set of arcs is partitioned into two separate sets of *input arcs* and *output arcs*. Each node may contain *tokens*. A token in a node may fire, consuming some tokens through input arcs and producing some other tokens through output arcs. So the same token may be considered as a passive resource (produced or consumed by an agent), or as an active agent (producing or consuming resources) at the same time.

In [5] AR-nets were extended to Resource Driven Automata Nets (RDA-nets) following the 'nets-within-nets' approach. Thus a RDA-net is a two-level model, where a system level of the RDA-net is defined by a flat AR-net, and agents are represented as finite state machines (automata), being placed into system nodes as a tokens. Agents interact through shared system resources, while agents themselves are shared system resources.

From the modeling perspective the system net in a RDA-net is a "map" of input and output ports (arcs), that connects different system nodes. The agent may use ports to interact with system resources (produce/consume), residing in adjacent nodes. So an agent may natively produce/consume/copy/move other agents through these ports (or even use itself as a resource).

In this paper we construct a native and transparent extension of RDA-nets to time dependent formalism using the time interval approach [16]. In timed RDA-nets (TRDA-nets) each agent (automata) transition is equipped with two time intervals: a waiting duration and a firing duration. Note, that TRDA-nets allow modeling both synchronous and asynchronous agent interaction, locality of resources in the Petri net style, agent mobility in the style of nested Petri nets, and agents producing/consuming other agents in the style of nets of active resources. So, in contrast to other known timed formalisms the number of simultaneously waiting/working agents is unlimited.

We demonstrate modelling facilities of the formalism and prove that TRDA-nets can be embedded (up to time-bisimulation equivalence) into Timed Automata, as well as into Time Petri nets, thus TRDA-nets integrate merits of both formalisms. We present also a prototype emulator for TRDA modeling, verification and performance analysis.

The paper is organized as follows. In Section 2 we recall the definitions of RDA-nets. In Section 3 timed RDA-nets are introduced. In Section 4 we compare TRDA-nets with Timed automata and Time Petri nets. We also present a software prototype for TRDA-net simulation-based performance analysis. Section 5 contains some conclusions.

2 Preliminaries

Let S be a finite set. A *multiset* m over a set S is a mapping $m : S \to Nat$, where Nat is the set of natural numbers (including zero), i. e. a multiset may contain several copies of the same element.

For two multisets m, m' we write $m \subseteq m'$ iff $\forall s \in S : m(s) \leq m'(s)$ (the inclusion relation). The sum and the union of two multisets m and m' are defined as usual: $\forall s \in S : (m+m')(s) = m(s)+m'(s),\ (m \cup m')(s) = max(m(s), m'(s))$.

By $\mathcal{M}(S)$ we denote the set of all multisets over S.

Let Ω be a finite set of *types*, *Const* be a finite set of typed individual objects. We call these objects constants. The type of a constant $c \in Const$ is denoted by $Type(c)$. For $a \in \Omega$ by $Const(a)$ we denote the set of all constants of type a. Define *Var* to be a set of variables typed with Ω-types.

Let then Π be a finite set of typed (by elements of Ω) *ports* (port names). The type of a port $\pi \in \Pi$ is denoted by $Type(\pi)$.

Informally, *Resource Driven Automata Net* (RDA-net) [5] is a finite oriented graph (a "system net") with two types of arcs (input and output), whose nodes are populated by finite communities of tokens ("agents", "resources" or simply "objects"). Each object is a finite automaton, whose transitions may consume other objects through input arcs and produce other objects through output arcs. So, the system level of an RDA-net is a *net of active resources* (AR-net) [3].

Definition 1. *A* system net *is a tuple* $SN = (V, I, O, \pi)$*, where* V *is a finite set of* resource nodes *(*vertices*);* $I : V \times V \to Nat$ *is a* consumption relation *(*input arcs*);* $O : V \times V \to Nat$ *is a* production relation *(*output arcs*);* $\pi : (I \cup O) \to \Pi$ *is a function, labeling arcs by port names.*

Graphically nodes are represented by circles, a consumption relation by dotted arrows and a production relation by solid arrows (Fig. 1).

Definition 2. *A* marking M *of the system net* SN *is a function* $M : V \to \mathcal{M}(Const)$*, that maps a multiset of objects to each node of the net.*

A marked system net *is a pair* (SN, M_0) *where* $SN = (V, I, O, \pi)$ *is a system net and* M_0 *is it's* initial marking.

We define a language L of *resource transformation expressions* as follows. Let $\pi \in \Pi$. An *input term* is a term of the form $\pi?e$, where e is a variable or a constant of type $Type(\pi)$. Similarly, an *output term* is a term of the form $\pi!e$ with the same conditions on π and e. Now, a resource transformation expression is a term of the form $\alpha_1; \alpha_2; \ldots; \alpha_k$, where α_j $(j = 1, \ldots, k)$ is either an input or an output term (types of subexpressions $\alpha_1, \alpha_2, \ldots, \alpha_k$ may differ).

Definition 3. *A* resource driven automaton *(RDA) is a tuple $A = (S_A, T_A, l_A)$, where S_A is a finite set of states, $T_A \subseteq S_A \times S_A$ is a transition relation, and $l_A : T_A \to L$ is a transition labeling.*

Thus a RDA is just a finite state machine with labeled transitions and no distinguished initial state. Further resource driven automata will play the role of tokens in system nets. In a special case, when a RDA contains exactly one state with no transitions, such a token is just a usual colored token without its own states and behavior. We call such tokens elementary resources.

Definition 4. *Let $\Omega = \{a_1, \ldots, a_k\}$ be a finite set of types, and $\mathcal{A} = (A_1, \ldots, A_k)$ be a finite set of RDAs, where*

1. *for a type $a_i \in \Omega$ the set $Const(a_i)$ is defined as the set of all states of RDA A_i; $Const =_{def} \bigcup_{a \in \Omega} Const(a)$;*
2. *each A_i is a RDA over the set of types Ω, the set Var of variables typed with Ω-types, and the set of constants $Const$.*

An RDA-net $RN = (\Omega, SN, (A_1, \ldots, A_k))$ *consists of a finite set of RDAs $(A_1, \ldots, A_k)$ with types from Ω as described above, and a system net SN over a set Ω of types, set $Const$ of constants, and some set Π of Ω-typed ports.*

A marking *(a state) in a RDA-net is a marking in underlying system net.*

By abuse of notation we will not differ automaton A and its name/type a. We will write $(a|s)$ for the constant denoting the state s of an automaton A of type a. Contextually we call a constant an *agent*, a *resource*, or just an *object*.

We define an interleaving semantics for RDA-nets. A run in a RDA-net is a sequence of agent transition firings. Thus only agents may change a state.

Informally speaking, firing a transition t in an agent a requires resources listed in input subterms of the resource transformation expression, labeling t. Input subterm $p?e$ describes a resource e, which should be obtained via a port p in the system net. A source node for this port arrow in a system net is a node, where this resource should be taken from. Similarly, if possible a firing of an automaton transition consumes required resources and produces new resources in line with output subexpressions of the transition label.

Definition 5. *A* binding *of the variables is a function* **bind** : $Var \to Const$ *such that for all $\varphi \in Var$ we have $Type(\mathbf{bind}(\varphi)) = Type(\varphi)$.*

Let M be a marking in a RDA-net $RN = (\Omega, SN, (A_1, \ldots, A_k))$, $a|s$ – a RDA in a state s residing in a node v of RN in M. Let b be a variable binding. A transition $t = (s, s')$ with a label $l_a(t)$ is *enabled* in M iff there is a one-to-one correspondence between input subterms in $l_a(t)$ and objects in M s.t. for a subterm $p?e$ there is an object $e[b]$ in some node v' of RN in M with an input arrow $(v', v) \in I$ labeled by p in RN (here $e[b]$ denotes e if e is a constant and $b(e)$ if e is a variable).

The correspondence described above defines a 'submarking' $\check{M}$ mapping a node v of RN to the multiset of objects residing in v and singled out by input subterms of $l_a(t)$. It includes all objects, consumed by firing of t with binding b.

Note, that for given t and b the marking $\check{M}$ is defined nondeterministically. By $\mathbf{in}(t[b])$ we will denote the set of all such markings.

Similarly we define a set $\mathbf{out}(t[b])$ of markings mapping a node v to objects produced by transition t with a binding b in line with output subterms of $l_a(t)$.

Definition 6. *Let $(A|s)$ be an agent, residing in a node $v \in V$ in a system marking M, and $t \in T_A$ be a transition with $t = (s, s')$.*

The transition t is enabled *with a variable binding b iff there exists a marking $\check{M} \in \mathbf{in}(t[b])$ s.t. for each $u \in V : \check{M}(u) \subseteq M(u)$.*

An enabled binded transition may (nondeterministically) fire to a new marking M' s.t. for each node $u \in V : M'(u) = M(u) - \check{M}(u) + \hat{M}(u)$, where $\check{M} \in \mathbf{in}(t[b])$ and $\hat{M} \in \mathbf{out}(t[b])$.

Let us consider a well-known Dining philosophers problem. This classical problem, being quite simple, comprises all basic elements of distributed systems.

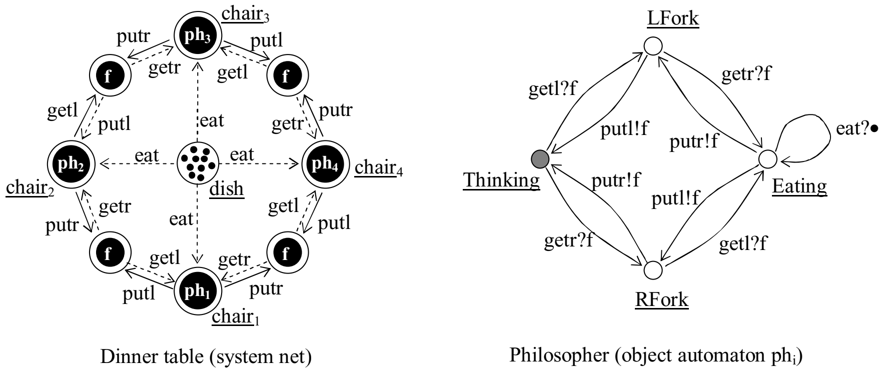


Fig. 1. Dining philosophers – Resource Driven Automata Nets representation

Philosophers (in our case, there are four of them) are sitting at a dinner table with a common <u>dish</u> of spaghetti in the middle. Between each of two philosophers there is a <u>fork</u> on the table. Spaghetti can be eaten only with two forks, so there is a potential conflict between neighbours. Each philosopher can be in one of four states: he is either <u>Thinking</u> (having no forks), <u>Eating</u> (having two forks), having a fork only in his right or left hand (<u>RFork</u> or <u>LFork</u>). In the state <u>Eating</u> the philosopher can also perform an action "eat" (take spaghetti).

Fig. 1 presents modeling Dining philosophers problem with RDA-nets. The left part of the picture shows the "spatial" distribution of places (chairs, forks and spaghetti) on a specific "map". The right part represents a behavior of a philosopher automaton. On the system level we specify types of relations between nodes by arc properties: a dashed arc denotes the consumption of a resource, a solid arc denotes the production. For example, dashed arcs "getl" ("get left") describe a possibility to take something from the left. Transition expressions define, what system resources should be produced or consumed by an agent.

For example, `putr?f` means that a fork should be put to the right, `eat?•` means that a simple token (a spaghetti) should be consumed through the port "eat".

A formalism of Resource Driven Automata Nets allows us to visually express concurrency in terms of autonomous (asynchronous) behavior, spatial distribution of objects (agents/resources) and sharing access to common resources. In [5] it was shown that RDA-nets are expressively equivalent to Petri nets, and hence crucial behavioral properties are decidable for them. At the same time this formalism allows natural and compact modeling of multi-agent systems with complex interactions. Moreover, it was shown [6] that RDA-nets can express sophisticated spatial dynamics up to cellular systems.

3 Timed RDA-Nets

Here we use a concept of timing constraints, introduced in [16] for Time Petri Nets (TPN). However, in our formalism not a single one but two different time intervals are associated with each transition. The first one is a *waiting duration interval.* An implicit clock can be associated with each enabled transition, and gives the elapsed time since it was last enabled. An enabled transition can fire if its clock value belongs to the waiting interval of the transition. The second one is a *firing duration interval.* Once again, an implicit clock is associated with each transition in the moment of its activating (we call this a moment of *starting*). A started transition already have consumed its input resources, but it must *stop* to produce its output resources and change the internal state of a corresponding object. The event of *stopping* occures nondeterministically, but its time must belong to the firing duration interval.

Let $\mathbb{R}_{\geq 0}$ denote non-negative real numbers. By *Interv* we denote a set of time intervals: $Interv =_{\text{def}} \{[x;y] \mid x \in \mathbb{R}_{\geq 0},\ y \in (\mathbb{R}_{\geq 0} \cup \{\infty\}) \text{ s.t. } x \leq y\}$.

Definition 7. *A* timed resource driven automaton *(TRDA) is a tuple* $A = (S_A, T_A, l_A, W_A, F_A)$*, where* (S_A, T_A, l_A) *is an RDA,* $W_A : T_A \to Interv$ *is a transition labelling by waiting duration intervals,* $F_A : T_A \to Interv$ *is a transition labelling by firing duration intervals.*

At any moment there is exactly one element of a TRDA (state or transition), marked by a time value (elapsed waiting time for a state or elapsed firing time for a trasition). All other elements of the automaton are idle and hence not time-marked. The timed run consists of alternating waiting and firing intervals. So there are two types of continuous timed states — (static) waiting and (progress) firing, and two types of discrete events — starting and stopping of transitions. This applies both to a single object automaton and to a net as a whole.

Definition 8. *Let A be a timed resource driven automaton. A set of* waiting constants *of type A (timed* waiting states*) is defined as:*

$$WConst(A) =_{def} \{(s,\nu) \mid s \in S_A, \nu \in \mathbb{R}_{\geq 0} \ s.t.\ (\exists t \in T_A : s \xrightarrow{t} s' \wedge \nu \in W_A(t))\};$$

Firing constants *of type A (timed variable-binded* firing states*) are defined as:*

$$FConst(A) =_{def} \{(t[b],\nu) \mid t \in T_A,\ b \text{ is a binding},\ \nu \in \mathbb{R}_{\geq 0}\ \ s.t.\ \ \nu \in F_A(t)\}.$$

Definition 9. *Let $\Omega = \{A_1, \ldots, A_k\}$ be a finite set of types. The sets of* waiting *and* firing constants *WConst and FConst are defined as*

$$WConst =_{def} \bigcup_{A \in \Omega} WConst(A) \quad and \quad FConst =_{def} \bigcup_{A \in \Omega} FConst(A).$$

The set of all constants (all timed automata states) is defined as

$$Const =_{def} WConst \cup FConst.$$

The formal definition of TRDA-net differs from the definition of RDA-net only by using timed constants instead of untimed:

Definition 10. A TRDA-net $RN = (\Omega, SN, (A_1, \ldots, A_k))$ *consists of a finite set of TRDAs $(A_1, \ldots, A_k)$ with types from Ω and a system net SN over a set Ω of types, set Const of constants, and some set Π of Ω-typed ports.*

A marking *(a state) in a TRDA-net is a marking in its underlying system net by constants from Const.*

Definitions of variable binding **bind** and binded marking sets $\mathbf{in}(t[b])$ and $\mathbf{out}(t[b])$ are exactly the same as for RDA-nets (but note that the corresponding sets of bindings and markings become continuous).

Definition 11. *(starting) Let $\big(A|(s,\nu)\big)$ be a waiting agent, residing in a node $v \in V$ in a system marking M, $t \in T_A$ be a transition with $t = (s, s')$, and $\nu \in \mathbb{R}_{\geq 0}$ be a time value.*

The transition t is enabled *with a variable binding b iff $\nu \in W_A(t)$ and there exists a marking $\check{M} \in \mathbf{in}(t[b])$ s.t. for each $u \in V : \check{M}(u) \subseteq M(u)$.*

An enabled binded transition may (nondeterministically) fire to a new marking M' (denoted $M \overset{start(t[b])}{\longrightarrow} M'$), such that for each node $u \in V$ with $u \neq v$ we have $M'(u) = M(u) - \check{M}(u)$, and for the node v we have:

1. $\check{M}(v)(s,\nu) = 0 \;\Rightarrow\; M'(v) = M(v) - \check{M}(v) - (s,\nu) + (t[b], 0)$;
2. $M(v)(s,\nu) = \check{M}(v)(s,\nu) = 1 \;\Rightarrow\; M'(v) = M(v) - \check{M}(v)$;
3. $M(v)(s,\nu) > \check{M}(v)(s,\nu) > 0 \;\Rightarrow$ *nondeterministically (1) or (2) is chosen.*

In situation (1) the agent starts, in situation (2) the agent removes itself.

Definition 12. *(stopping) Let $\big(A|(t[b],\nu)\big)$ be a firing agent, residing in a node $v \in V$ in a system marking M, $t \in T_A$ be a transition with $t = (s, s')$, b be a variables binding, and $\nu \in \mathbb{R}_{\geq 0}$ be a time value.*

The binded transition $t[b]$ is ready to stop *iff $\nu \in F_A(t)$.*

A ready to stop binded transition may (nondeterministically) stop, changing M to a new marking M' (denoted $M \overset{stop(t[b])}{\longrightarrow} M'$), such that for each node $u \in V$ with $u \neq v$ we have $M'(u) = M(u) + \hat{M}(u)$, and for the node v we have $M'(v) = M(v) + \hat{M}(v) - (t[b], \nu) + (s', 0)$.

Definition 13. *(time elapsing) Let M be a system marking. A time interval $\mu \in \mathbb{R}_{\geq 0}$ may* pass, *changing M to some marking M' (denoted $M \overset{\mu}{\to} M'$), if for each system node $u \in V$ and type $A \in \Omega$ we have:*

- *For each waiting constant* $(s,\nu) \in \mathit{WConst}(A)$, *s.t.* $M(u)(s,\nu) > 0$, *we have* $(s,\nu+\mu) \in \mathit{WConst}(A)$, *and, moreover, there exists no enabled transition* $t[b]$ *with* $t = (s,s')$ *for some* $s' \in S_A$, *such that* $\nu \in W_A(t)$ *and* $\nu + \mu \notin W_A(t)$. *In this case let* $M'(u)(s,\nu) = 0$ *and* $M'(u)(s,\nu+\mu) = M(u)(s,\nu)$.
- *For each firing constant* $(t[b],\nu) \in \mathit{FConst}(A)$, *s.t.* $M(u)(t[b],\nu) > 0$, *we have* $(t[b],\nu+\mu) \in \mathit{FConst}(A)$. *In this case let* $M'(u)(t[b],\nu) = 0$ *and* $M'(u)(t[b],\nu+\mu) = M(u)(t[b],\nu)$.

Note that if the marking of the system net contains some waiting automaton $\big(A|(s,\nu)\big)$, such that at least one of its enabled transitions can not wait any more (i.e. $W(t) = [x,\nu]$), then no time elapsing can happen. Before any continuous step there have to occure some discrete event — either a transition t starting, or a transition starting in some other automaton, consuming $\big(A|(s,\nu)\big)$. In other words, we implement a most widely used strong time semantics (STS), also known as "urgency policy".

Also note that similar observation is true for transitions: each firing must be ended (either stopped by itself or interrupted by some other automaton). Hence, in TRDA-nets the strong time semantics applies to both static and progress projections of a model.

And the final remark is on objects, simultaneously consumed and produced through a binded variables. Assume that a transition t is labelled by a resource expression $(in?x; out!x)$, containing some variable x of some type A. This variable can be binded to any constant of type A, even to a waiting or firing automaton. But while the transition t is firing, this automaton is "removed" from the model, and hence its internal time is not elapsing. In other words, transition "freezes" all the related resources in the process of firing.

We illustrate our formalism by a well-known concurrency use case — Cleaners Problem. The problem is usually stated as follows: there are three mutually connected rooms, each room has a table and a wastepaper basket. In the room there is also a "paper generator" — some worker, that can produce both good papers and garbage. The organization also has three cleaners. A cleaner can put a paper on a table and a garbage to the basket. Moreover, he can move to another room (we assume that he visits all rooms iteratively in a strict order).

The model is represented on Fig. 2. The "map" of the first room is given in details, the two other room have the same structure. We use four types of tokens: simple resources `paper` and `trash`, and two active agents — `Generator` and `Cleaner`. `Generator` has a single state and three transitions — `paper` (produce paper), `garbage` (produce garbage) and `wait` (do nothing). `Cleaner` has similar one-state structure, but the set of transitions is different — `clean` (take garbage and put it to the basket), `store` (take paper and put it on the table) and `go` (move to the next room).

The graphical structure of the system level of the model is quite convenient and allows to observe both (logical) parallelism and (spatial) movement. The resources (papers) are incorporated seamlessly, since visually they are tokens — just like more complex active agents (people). Continuous part of the model is concentrated on the object level — timing constraints are imposed on the

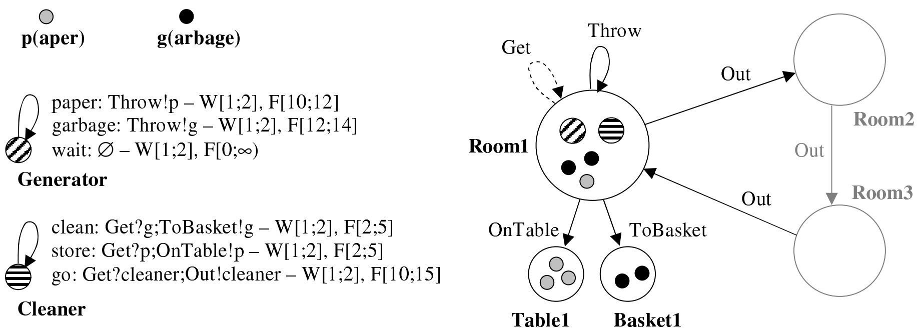

p(aper)
g(arbage)
paper: Throw!p – W[1;2], F[10;12]
garbage: Throw!g – W[1;2], F[12;14]
wait: ∅ – W[1;2], F[0;∞)
Generator
clean: Get?g;ToBasket!g – W[1;2], F[2;5]
store: Get?p;OnTable!p – W[1;2], F[2;5]
go: Get?cleaner;Out!cleaner – W[1;2], F[10;15]
Cleaner
Get
Throw
Out
Room1
Room2
Out
Room3
Out
OnTable
ToBasket
Table1
Basket1

Fig. 2. "Cleaners" (first room fragment)

transitions. For example, we know that `Cleaner` needs at least 10 and at most 15 time units to move from one room to another.

Theorem 1. *TRDA-nets are expressively equivalent to Turing machines. Moreover, the following four special cases of TRDA-nets are Turing-powerful:*
For any TRDA $A = (S_A, T_A, l_A, W_A, F_A)$ and transition $t \in T_A$ we have

- $W_A(t) = [x; x]$ *and* $F_A(t) = [y; y]$ *for some* x, y *(constant waiting and firing);*
- $W_A(t) = [x; x]$ *and* $F_A(t) = [y; \infty)$ *for some* x, y *(constant waiting and unlimited firing);*
- $W_A(t) = [0; 0]$ *and* $F_A(t) = [x; x]$ *for some* x *(no waiting and const. firing).*

The proof is based on the constructing of an equivalent TRDA for any given Minsky machine. We omit it due to lack of space.

4 Modelling and Verification with TRDA-Nets

In this subsection we consider two classical dense time formalisms — Timed Automata and Time Petri Nets, and by comparison estimate the TRDA-nets modelling and verification pros and cons.

Timed Automata. Timed automata (TA) were introduced by Alur and Dill [1,2] and have by now been recognized as one of the classical formalisms for modelling sequential real-time systems with dense time. A timed automaton is a finite-state machine extended with a finite number of synchronous clocks. Transitions in the automaton are conditioned on the clock values and taking a transition can reset the values of selected clocks.

A TRDA can be effectively translated into time-bisimulation-equivalent TA (Algorithm 1 in the Appendix). This translation allows to use a wide range of elaborated tools for TRDA verification on the object level (such as UPPAAL, KRONOS, IF, CMC and others).

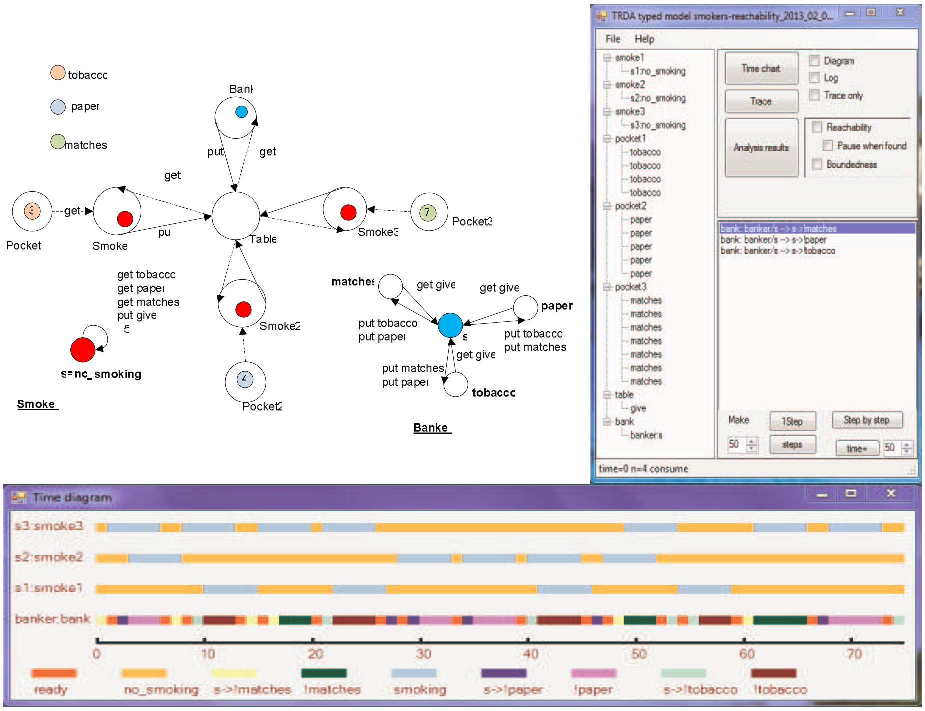
Time Petri Nets. In Time Petri Nets (TPN), introduced by Merlin [16], each transition has an associated time interval which gives the earliest and latest firing time of the transition since it became enabled. The transition can fire as soon as the clock value reaches the earliest firing time and it must fire no later than the latest firing time, unless the transition got disabled by some other transition.

Unlike TA for *sequential* systems, TPN is not the only undisputed formalism for *distributed* systems. However, the model of TPN has been around for several decades and it has proven to be useful for modelling of a wide range of distributed real-time systems including workflow processes, scheduling problems and others. There are available a few verification tools like TINA and ROMEO (mostly useful for bounded TPN, since unbounded TPN have full Turing power).

Both TRDA and TRDA-net can be effectively translated into time-bisimulation-equivalent TPN (Algorithms 2 and 3 in the Appendix). Thus we can use an effective combination of TRDA modeling and TPN verification/model checking.

It can be easily seen that TRDA-nets have features of both TA and TPN. Like in TA, the elementary agent is a sequential system — an automaton. This allows simple modelling of dense time transition diagrams. Like in TPN, we can compose subsystems in a distributed resource-constrained net. This allows simple modelling of parallelism and synchronization.

As we have shown, TRDA-nets may be a convenient modeling formalism for distributed dense time systems. In this section we present a prototype TRDA-based software tool, that allows some specific model checking and visualization.



smoke1
s1:no_smoking
smoke2
s2:no_smoking
smoke3
s3:no_smoking
pocket1
tobacco
tobacco
tobacco
tobacco
pocket2
paper
paper
paper
paper
paper
pocket3
matches
matches
matches
matches
matches
matches
matches
table
give
bank
banker:s

```
bank: banker/s --> s->!matches
bank: banker/s --> s->!paper
bank: banker/s --> s->!tobacco
```

Make 50 steps; time+ 50

time=0 n=4 consume

Fig. 3. "Smokers" — TRDA+ emulator

Consider a specific use case — Cigarette Smokers problems. This is a well-known concurrency problem in computer science, originally described in [18].

There is a table near which three smokers are seated. One smoker has a number of tobacco resource in his pocket, the second one — smoking paper, the third one — matches. And there is a banker, who put two different components of cigarette on the table, when there is nothing on the table. The smoker who has the third complementary resource — starts smoking.

An example of modelling and model-checking is given on Fig. 3. Here we used a number of features of our tool to check some properties of dense-time behaviour of the considered system. The following behavioral properties can be automatically analyzed (modulo finite execution traces): liveness, liveness of system net transitions, k-boundedness, reachability, deadlock freedom.

The following statistical analysis is also possible: average/total waiting, average/total firing times, number of visits. The tool works with a specific TRDA-representation language, allowing to load a model into model-checker.

5 Conclusion

In this paper we have presented a new formalism of TRDA-nets and a tool for modeling time dependent distributed systems of agents, communicating via resource consuming/production. Being Turing powerful, TRDA-nets inherit features and merits of both Timed Automata and Time Petri Nets. Rich modelling facilities of TRDA-nets were illustrated by some classical case studies.

In the further work we plan to study some important semantical (bounded, live) and syntactical (free-choice) subclasses of TRDA-nets. Considering RDA-nets with Timed automata as net tokens is another interesting challenge, as well as connecting TRDA-nets with other dense time formalisms, such as Timed Petri nets [19] and Timed-Arc Petri nets [8].

References

1. Alur, R., Dill, D.: Automata for modelling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
2. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Bashkin, V.A.: Nets of active resources for distributed systems modeling. Joint Bulletin of NCC&IIS, Comp. Science 28, 43–54 (2008)
4. Bashkin, V.A.: Formalization of semantics of systems with unreliable agents by means of nets of active resources. Progr. and Comp. Soft. 36(4), 187–196 (2010)
5. Bashkin, V.A., Lomazova, I.A.: Resource Driven Automata Nets. Fund. Inf. 109(3), 223–236 (2011)
6. Bashkin, V.A., Lomazova, I.A.: Cellular Resource Driven Automata Nets. Fund. Inf. 120(3-4), 245–259 (2012)
7. Bednarczyk, M.A., Bernardinello, L., Pawłowski, W., Pomello, L.: Modelling mobility with Petri Hypernets. In: Fiadeiro, J.L., Mosses, P.D., Orejas, F. (eds.) WADT 2004. LNCS, vol. 3423, pp. 28–44. Springer, Heidelberg (2005)

8. Bolognesi, T., Lucidi, F., Trigila, S.: From timed Petri nets to timed LOTOS. In: Proc. of the IFIP WG 6.1 Tenth International Symposium on Protocol Specification, Testing and Verification, pp. 1–14 (1990)
9. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983)
10. Chang, L., He, X., Lian, J., Shatz, S.: Applying a Nested Petri Net Modeling Paradigm to Coordination of Sensor Networks with Mobile Agents. In: Proc. of Workshop on Petri Nets and Distributed Systems 2008, Xian, pp. 132–145 (2008)
11. Dworzanski, L.W., Lomazova, I.A.: On Compositionality of Boundedness and Liveness for Nested Petri Nets. Fund. Inf. 120(3-4), 243–257 (2012)
12. Köhler-Bußmeier, M.: Hornets: Nets within Nets combined with Net Algebra. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 243–262. Springer, Heidelberg (2009)
13. Lomazova, I.A.: Nested Petri Nets – a Formalism for Specification and Verification of Multi-Agent Distributed Systems. Fund. Inf. 43(1-4), 195–214 (2000)
14. Lomazova, I.A.: Communities of Interacting Automata for Modelling Distributed Systems with Dynamic Structure. Fund. Inf. 60(1-4), 225–235 (2004)
15. Lomazova, I.A.: Nested Petri nets for adaptive process modeling. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 460–474. Springer, Heidelberg (2008)
16. Merlin, P.M.: A Study of the Recoverability of Computing Systems. PhD thesis, University of California, Irvine, CA, USA (1974)
17. Nehaniv, C.L.: Asynchronous Automata Networks Can Emulate Any Synchronous Automata Network. Int. J. of Algebra and Computation 14(5-6), 719–739 (2004)
18. Patil, S.S.: Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes. MIT, Project MAC, CSG Memo 57 (1971)
19. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. Technical report. Massachusetts Institute of Technology, Cambridge (1974)
20. Valk, R.: Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–24. Springer, Heidelberg (1998)

Appendix

Algorithm 1. (Translation of a TRDA to a Timed Automaton) Here we show that an internal timed transition diagram of a TRDA (without resource expressions) can be reperesented as a Timed Automaton [1,2]. It is sufficient to present a transformation of a single transition.

Consider a transition $s \xrightarrow{W[a;b];F[c;d]} s'$. In the resulting TA it will be modelled by three locations (l_s, l'_s and l_t) and two transitions ($l_s \xrightarrow{t_{start}} l_t$ and $l_t \xrightarrow{t_{stop}} l'_s$). Additionally we introduce two clocks — $wait_s$ for the state s and $fire_t$ for the transition t. Transition t_{start} is labelled by guard expression $wait_s \geq a \wedge wait_s \leq b$ and reset expression $fire_t := 0$. Transition t_{stop} is labelled by guard expression $fire_t \geq c \wedge fire_t \leq d$ and reset expression $wait_{s'} := 0$.

This is a syntactical transformation, and the resulting TA has exactly the same structure and timed behaviour as the TRDA (modulo resources).

Algorithm 2. (Translation of a TRDA to a Time Petri Net) Here we show that a TRDA (with resource expressions) can be reperesented as a TPN.

First consider a transition diagram of TRDA. It is replaced by an automaton Petri net (where each transition of a Petri net has one input and one output place) of the same graph topology, but with "doubled" transitions — each transition t of TRDA is replaced in TPN by a sequence of two transitions — t_{start} and t_{stop}. Assuming $W(t) = [a; b]$ and $F(t) = [c; d]$, we set the firing interval of t_{start} to $[a; b]$, and the firing interval of t_{stop} to $[c; d]$.

At this stage the resulting TPN behaves exactly like an internal timed transition diagram of a given TRDA (without resource expressions). Now consider all resource expressions. Let $\Omega = \{A_1, \dots, A_k\}$ be the set of all types (the set of all possible "colours" of resource tokens). We add to the net so-called "resource places" — one for every type — denoted by $p_{A_1}, \dots, p_{A_k}$.

And the final step: if the original TRDA transition t is labeled by a resource expression $\pi?a$ of type A_i, then we add an arc from place p_{A_i} to transition t_{start}, if it is labelled by a resource expression $\pi!a$ of type A_i, then we add an arc from t_{stop} to p_{A_i}. Like the previous algorithm, this is just a syntactical transformation.

Algorithm 3. (Approximation of a TRDA-net by a Time Petri Net) Here we show that a whole TRDA-net (with resource expressions) can be reperesented as a Time Petri Net. However, the modeling is not exact (in the sence of dense time simulations) — and it cannot be exact since in TRDA-net the number of simultaneously firing transitions is unbounded (we can produce any number of TRDA tokens). In TPN the number of transitions is fixed, hence we can model TRDA-net by a TPN only in a weaker sense — considering some kind of approximation of TRDA dense time state space.

The complete transformation uses the method, introduced in the previous algorithm. However, now we construct a set of TPN — one net for each pair (TRDA type, system node), and a set of resource places — one place for each pair (TRDA type, system node). The resource-consuming and resource-producing arcs are introduced just like in the previous algorithm.

Parallelization Properties of Preconditioners for the Conjugate Gradient Methods

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 the analysis of parallelization properties of several typical preconditioners for the Conjugate Gradient methods. For implicit preconditioners, geometric and algebraic parallelization approaches are discussed. Additionally, different optimization techniques are suggested. Some implementation details are given for each method. Finally, parallel performance results are presented and discussed.

1 Introduction

Conjugate Gradient methods are widely used for solving large linear systems arising in discretizations of partial differential equations in many areas (fluid dynamics, semiconductor devices, quantum problems). They can be applied to ill-conditioned linear systems, both symmetric (plain CG) and non-symmetric (BiCGStab, GMRES etc.). In order to accelerate convergence, these methods require preconditioning. Now, with the proliferation of multicore and manycore processors, efficient parallelization of preconditioners becomes very important.

There are two main classes of preconditioners: explicit, that apply only a matrix-vector multiplication, and implicit, that require solution of auxiliary linear systems based on the incomplete decomposition of the original matrix. Explicit preconditioners act locally by means of a stencil of limited size and propagate information through the domain with low speed, while implicit preconditioners operate globally and propagate information instantly. Due to this implicit preconditioners work much faster and have better than linear dependence of convergence on the geometric size of the problem.

Parallel properties of preconditioners strongly depend on how information is propagated in the algorithm. For this reason implicit preconditioners can't be easily parallelized, and many efforts are needed for finding geometric and algebraic approached of parallelization. There exists a separate class of implicit methods, Multigrid, which possesses very good convergence and parallelization properties. However, Multigrid is extremely difficult for implementation, and in some cases it can't be applied at all. Due to this classical (explicit and implicit) preconditioners are still widely used in many numerical applications.

Thereby, in this paper we will analyze parallelization properties and performance of several preconditioners for different discretizations and geometries, and their implementation details on modern multicore processors.

2 Conjugate Gradient and Preconditioners

The original non-preconditioned Conjugate Gradient method [1] of the solution of a linear system $A\boldsymbol{x} = \boldsymbol{b}$ is very simple for implementation and can be easily parallelized. However, because of the explicit nature, it has low convergence rate. Because of this, the CG method is usually applied to the preconditioned linear system $(M^{-1}A)\boldsymbol{x} = M^{-1}\boldsymbol{b}$ where M is a symmetric positive-definite matrix that is "close" to the main matrix A (which is also symmetric and positive-definite).

Preconditioning works well if the condition number of the matrix $M^{-1}A$ is much less than that of the original matrix A. The simplest way to reduce this condition number and accelerate the convergence is to apply an "explicit" preconditioner ($B = M^{-1}$) than doesn't require the inversion of M.

A good example of this sort is the polynomial Jacobi preconditioner that is based on a truncated series of the approximation $1/(1-a) = 1 + a + a^2 + \ldots$

$$B = M^{-1} = \sum_{k=0}^{n} (H^k)P^{-1} \text{ where } P = \mathrm{diag}(A),\ H = P^{-1}(P-A) = I - P^{-1}A$$

For $n = 0$, this expression represents the diagonal preconditioner $B = P^{-1}$ (not considered as a true preconditioner because of its simplicity). For $n = 1$, it looks as $B = (I + (I - P^{-1}A))P^{-1}$ and improves acceleration rate by two times (with some increase of computational cost). This corresponds to the expansion of the computational stencil of one iteration of the algorithm. Therefore, it can be easily applied and parallelized. Variants for $n = 2$ or $n = 3$ are more complex and not enough efficient, for this reason they are not considered here.

Neither sort of the simple explicit preconditioner can improve the convergence radically. For this reason, it is desirable to use implicit preconditioners. The most popular implicit preconditioner is Incomplete LU (ILU) decomposition [2]. To be efficient, this decomposition must satisfy the following conditions:

- Preconditioner matrix M must be chosen "close" to the main matrix A in such a way that (for typical values of vector $\boldsymbol{x}$) the approximation error is sufficiently small: $||M\boldsymbol{x} - A\boldsymbol{x}|| = \varepsilon\, ||\boldsymbol{x}|| \ll ||\boldsymbol{x}||$, or $\varepsilon \ll 1$.
- Matrix M must be suitable for decomposition into factors (e.g. $M = LU$) and these factors must be invertible with low computational cost, i.e. must allow economical solution of auxiliary linear systems $L\boldsymbol{y} = \boldsymbol{z}$ and $U\boldsymbol{x} = \boldsymbol{y}$.
- Solution of these linear systems must be subject to efficient parallelization.

Straightforward implementation of ILU preconditioner doesn't approximate the main matrix A with the required accuracy, i.e. $\varepsilon = O(1)$ [3]. As the result, its convergence properties are not good: $O(N)$ iterations are required as for explicit preconditioners (N is the dimension of a problem in one spatial direction), though the total number of iterations may become less. More accurate Modified ILU (MILU) preconditioner approximates the main matrix A with the accuracy $\varepsilon = O(h)$ (here h is the grid distance) resulting in $O(N^{\frac{1}{2}})$ iterations [4,3]. However, Modified ILU can't be accurately applied in some situations (e.g. for solving systems of equations and for the domain decomposition approach [2]).

On the other hand, both ILU and MILU can't be massively parallelized because of recursive nature of forward and backward sweeps ($\boldsymbol{Ly} = \boldsymbol{z}$, $\boldsymbol{Ux} = \boldsymbol{y}$). One idea is to use red-black grid numbering in order to parallelize sweeps. However, in this case ILU decomposition looses its implicit properties and demonstrated $O(N)$ behaviour. Other sorts of complicated explicit preconditioners (like approximate inverse) also propagate information slowly because of the limited stencil size and also belong to the $O(N)$ class.

For these reasons, simple explicit preconditioners remain attractive in some cases because of good parallelization properties. They will be considered in the next section, followed by the analysis of two sorts of implicit preconditioners: Modified ILU for Cartesian discretization in a regular domain, and plain ILU for general sparse matrices.

3 Optimization and Parallelization of Explicit Preconditioners

For the analysis of simple explicit preconditioners, solution of linear systems arising in the discretization of the Navier-Stokes equation is considered. For the components of velocity (u, v, w) these are non-symmetric linear systems to be solved by the BiCGStab, for pressure (p) the plain Conjugate Gradient method can be used. Sparse matrices of these linear systems are stored in the Compressed Row Storage (CRS) format. The test problem uses the Cartesian discretization within a spherical domain, number of grid points is about 320000. The algorithm is parallelized for shared-memory computer systems using OpenMP [5]. This method can be extended to the hybrid OpenMP/MPI environment.

The Polynomial Jacobi preconditioner of the 1-st order is used both in BiCGStab and CG solvers. If the original matrix is diagonally scaled: $\mathrm{diag}(A){=}I$, the preconditioner will look as $B = 2\,I - A$. For the stable work, this preconditioner should be slightly underrelaxed by the following way: $B = I + \gamma\,(I - A)$. In the current implementation $\gamma = 0.985$.

The main computational kernel of both Conjugate gradient and preconditioning algorithms is the multiplication of a sparse matrix by a vector. Parallelization of this kernel in OpenMP is straightforward: the matrix is split into parts with equal number of rows, and each processor core independently computes the corresponding part of the resulting vector.

Parallel performance of such kind of computations is limited by the ability of the memory subsystem to read (write) data with the high speed. Therefore, data access rate requirements of the algorithm should be reduced. The Polynomial Jacobi preconditioner can be improved in this respect by using the single precision format (`real*4`) for storing a copy of the main matrix A for the preconditioning operator. Convergence properties of this preconditioner remain unchanged.

For the symmetric matrix, the CRS format is not convenient because it is not necessary to store its upper part. It is more economical and efficient to store only the strictly lower part L_A of the matrix A. If A is diagonally scaled, its representation will look as $A = L_A + I + L_A^{\mathrm{T}}$.

Multiplication of the symmetrically stored sparse matrix by a vector is more complex in comparison with the original storage scheme. The lower (L_A) and the upper (L_A^{T}) parts of the matrix are multiplied by a vector in different ways:

– for the lower part, scalar product of the densely packed row by the sparsely distributed elements of a vector is calculated, and the corresponding element of the resulted vector is modified;
– for the upper part, elements of the packed column are multiplied by the corresponding scalar value of a vector, and the sparsely distributed elements of the resulting vector are modified.

Parallelization of the symmetric sparse matrix multiplication algorithm is not straightforward. Unlike the original algorithm, the new one will not work correctly if we simply split all arrays by equal parts. As illustrated on Fig. 1 (left), parallel execution of the algorithm in different threads leads to the modification of the same elements of the resulting vector: thread 0 can modify elements in the data area of the immediately preceding thread 1 and corrupt its results.

In order to avoid this problem, the multicolored (or red-black) partitioning of data arrays can be applied. If we first split all arrays by equal parts in accordance with the number of threads, and then additionally split each part by two subparts of different color (marked as 0 and 1 on Fig. 1, left), we will be able to perform computations simultaneously in all subparts of the same color. After finishing processing for the particular color, we will do the same for another color. This approach is conceptually similar to the multicolored grid partitioning for some iterative methods (such as Gauss-Seidel and SOR).

The above variant of the multicolored partitioning has the obvious limitation: the maximal half-width of the matrix must be less than the size of the subpart otherwise a particular thread would modify the resulting vector of the preceding thread of the same color. In fact, this is a limitation on the number of threads for a given matrix. For the current example, this limitation is equal to 28, that is more than the number of processor cores in most available multiprocessor servers (usually 12 to 16). With increasing the problem size, this limit also will be

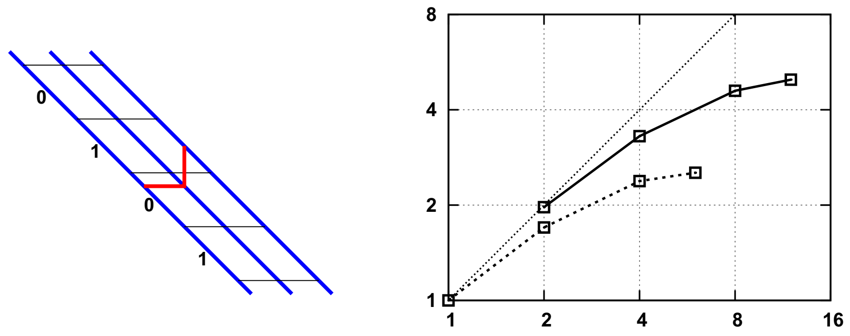


Fig. 1. Parallelization for the symmetric storage scheme (left). Parallelization results: dashed line – one processor, solid line – two processors (right)

increased. For long domains, it is recommended that grid nodes are enumerated in the direction of the short dimensions. In this case, the matrix bandwidth will be lower, and the thread limit will increase. Also, reducing the matrix width is useful from the performance point of view. However, if massive parallelization becomes necessary, it will be possible to develop more complex multicoloring approach similar to the multicolored grid partitioning.

Parallelization efficiency results for the above problem are shown on Fig. 1 (right). These results were obtained on the system with two 6-core Intel Xeon X5650 processors. Each processor has its own integrated 3-channel memory controller, therefore the peak memory access rate of the system is doubled. Parallelization tests were performed with 1 to 6 threads on one processor, and with 2 to 12 threads on two processors.

The one-processor results on Fig. 1 (right) demonstrate that the memory subsystem of a processor is almost saturated with 4 threads (additional performance increase with 6 threads is only about 6%). Similar saturation can be seen on the two-processor results. On the other hand, the two-processor results demonstrate almost linear performance increase in comparison to the single-processor runs with the same number of threads per processor (by about 1.95 times).

Thus, this test program is strongly memory-bound. The principal factor that limits parallel performance of such jobs is the memory throughput rate, while the CPU frequency is less important. The new generation Intel Xeon servers based on Sandy Bridge EP processors with 4 memory channels have much higher memory speed (51.2 GB/s vs 32 GB/s). As a result, parallel performance of similar jobs is proportionally increased. In the near future, after the switch from DDR3 to DDR4, processors with even faster memory subsystems will appear.

The above results demonstrate that explicit preconditioners have high parallelization potential and good scalability. The main factor that limits performance of explicit preconditioners is the peak memory access rate. For multiprocessor and multinode (cluster) computer systems, this limitation is scaled with the number of processors, thus increasing the total performance potential.

4 Implicit Preconditioners for Regular Domains

In this section we consider parallelization of the efficient Modified ILU (MILU) preconditioner for solving Poisson equation in rectangular parallelepipedic domains. As mentioned above, forward and backward sweeps of incomplete decomposition algorithms are recursive in their nature and can't be straightforwardly parallelized. Therefore, it is necessary to find such geometric properties of the algorithm that parallelization would become possible and efficient.

The original idea is taken from the twisted factorization of a tridiagonal linear system, when Gauss elimination is performed from both sides (for a subdiagonal and a superdiagonal, respectively). This idea can be naturally generalized to 2 and 3 dimensions. This method is called "nested twisted" [6,3]. An example of the nested twisted factorization is shown on Fig. 2 for two-dimensional case.

The nested twisted factorization method can be used for direct parallelization of the solution for up to 8 threads (in a Cartesian domain). The computational

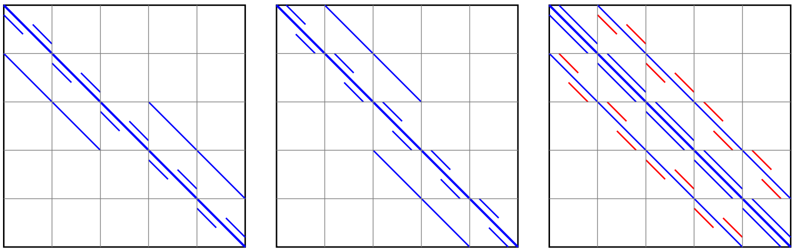


Fig. 2. Nested twisted factorization $L \cdot L^{\mathrm{T}} \to M$ suitable for parallelization

scheme of this method is as following. A rectangular parallelepipedic domain is split into 8 octants by separator planes (Fig. 3). In each octant, Gauss elimination is performed from the corner in the direction inwards (in all 3 dimensions), independently in different threads (Fig. 3, left).

After finishing eliminations in the internal points of octants, they are performed in quadrants of separator planes by the same way (Fig. 3, center). Then, intersection lines of separator planes are processed, and finally a solution at the central point is computed. The following backsubstitution is performed in the reverse order, from the central point in the direction outwards.

Parallelization for 16 threads needs another approach. For recursive algorithms, the staircase (or pipeline) method can be employed [7,3]. This method is illustrated on Fig. 3 (right). Each subdomain is split into 2 parts in the direction of `j` (see the bottom-left octant divided between threads 0 and 1). Computations in a plane `(i,j)` for a particular `k` can't be performed by thread 1 until they are finished by thread 0. However they can be fulfilled in a pipelined fashion: thread 1 computes a layer for some `k` at the same time when thread 0 computes the next layer for `k+1`. This method needs the synchronization between threads in a pair: before starting computations for some `k`, thread 1 must wait for thread 0 to finish computations in the same layer. At the backsubstitution stage of the algorithm, computations are performed in the reverse order.

Implementation of this method leads to some algorithmic overhead because at the beginning (for the first `k`) thread 1 is idle waiting for the results from thread 0, and at the end (for the last `k`) thread 0 is idle after finishing its work.

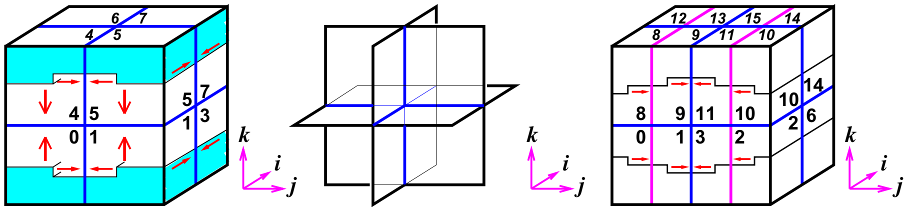


Fig. 3. Parallelization of the nested twisted factorization: illustration of the method (left); separator planes (center). Parallelization for 16 threads, staircase method (right)

The above method of parallelization can be used for more than two threads. In this case the algorithmic scheme will have more stairs and more points of synchronizations. However, because of performance overheads, the reasonable number of stairs can't be large. Therefore, parallelization potential of the above method can be estimated to be at most 32 or 64.

For computational domains of different shape, this potential depends on the number of corners. For example, a cylindrical domain has only 4 independent corners, and its parallelization potential will be at most 16 or 32. As a consequence, massive and efficient parallelization of MILU-class preconditioners for irregular domains and general sparse matrices is not possible at all.

The above parallelization method is "direct" with respect to the algebraic properties of the preconditioner matrix in such sense that the order of its approximation error remains the same as for the original (sequential) decomposition. For this reason, Modified ILU decomposition retains its convergence properties. This is not true, however, for the class of domain decomposition methods [2], when preconditioning accuracy is lost and convergence is sacrificed.

Parallelization efficiency results for the above method can be found in [3]. They are very similar to those in the previous section: the method demonstrates good efficiency if the memory subsystem is scaled with the number of threads, otherwise saturation is observed. Thus, this method is strongly memory-bound, and its performance depends firstly on the achievable memory throughput rate.

5 Implicit Preconditioners for Unstructured Grids

In this section we analyze parallelization approaches for Incomplete LU decomposition of general non-symmetric sparse matrices. These matrices are produced from the discretization of coupled systems of equations in irregular domains for the solution of stiff problems arising in different multiphysics applications (CFD, semiconductor transport, kinetic and quantum problems) [8,9].

Numerical solution of such ill-conditioned problems needs efficient CG-class solvers with ILU preconditioning. Here, parallelization of the preconditioned non-symmetric IDR solver will be considered [9]. The same ideas can be applied to any other CG-class solver for non-symmetric matrices (CGS, BiCGStab, GMRES).

Most computations in this solver are performed in two kernels – multiplication of the non-symmetric sparse matrix by a vector ($\boldsymbol{y} = A\boldsymbol{x}$), and solution of the decomposed linear system ($L\boldsymbol{y} = \boldsymbol{z}$, $U\boldsymbol{x} = \boldsymbol{y}$).

Parallelization of the first kernel is simple: if matrix A is stored in the CRS format, each thread can independently compute a part of the resulting vector. This is similar to the procedure described in the previous sections. This procedure has no strict limitations on the number of parallel threads.

On the other hand, procedure of Gauss elimination for a general sparse matrix can't be easily parallelized because such matrix does not possess any geometric parallelization properties. Thus it is necessary to look for algebraic approaches. The first idea is to use twisted factorization. An example of this factorization for a general banded sparse matrix is shown on Fig. 4.

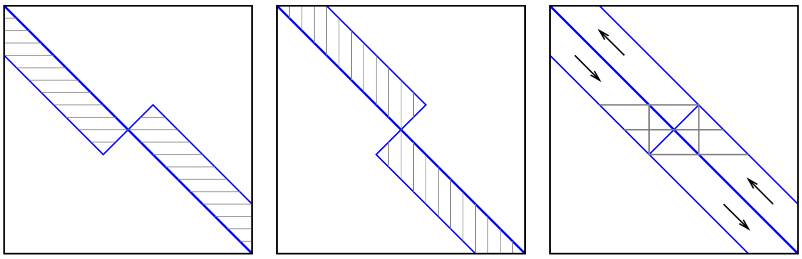


Fig. 4. Twisted factorization of the sparse matrix (left, center); illustration of the parallel Gauss elimination (right)

It can be seen that matrix factors have mutually symmetric portraits. These factors will be traditionally named as L and U. It is convenient to represent the decomposition as $M = (L + D)D^{-1}(D + U + R)$. Here R is the reverse diagonal that separates two part of each factor. The role of this reverse diagonal is seen on Fig. 4 (right) where the preconditioning matrix M is represented as the product of these factors. We can distinguish three areas on the matrix portrait: the central part (a square area with adjacent subareas on the left and on the right), the first part located above, and the last part located below.

Elimination of non-zero elements in the first and in the last parts is straightforward (as in the standard twisted-Gauss process) because these parts are formed by the simple multiplication of corresponding parts of factors L and U. Elimination of non-zero elements in the central part is much more complicated, because it is formed by the multiplication of complex central parts of L and U. Processing of the central part can't be parallelized and is performed serially. Parallel Gauss elimination is shown schematically on Fig. 4 (right).

This approach allows to parallelize the second kernel of the algorithm by 2 threads. It can be combined, for example, with 4-thread parallelization of the first kernel (multiplication of the main matrix by a vector) in order to achieve reasonable overall parallel efficiency on a multicore processor. This algorithm is also memory-bound (as those in the previous sections), and therefore its performance strongly depends on the throughput limitations of the memory subsystem.

Another parallelization approach for Gauss elimination is similar to the staircase (pipelined) method [7,3]. The idea of the approach is to split each matrix half-band (L and U) into pairs of adjacent trapezoidal blocks that have no mutual data dependences and therefore can be processed in parallel. As a result, parallelization of Gauss elimination for 4 threads will be implemented.

This new block-pipelined parallelization method is illustrated on Fig. 5. An example of the splitting of a matrix half-band is shown on Fig. 5 (center): the sub-diagonal part of L is split into adjacent pairs of blocks marked "0" and "1" (all other half-bands of matrices L and U are split into blocks similarly).

Let's consider the highlighted pair of blocks "0" (layer `k+1`) and "1" (layer `k`). It is seen that block "0" can be processed before finishing processing of block "1", provided the maximal column index of elements in "0" is less that the index

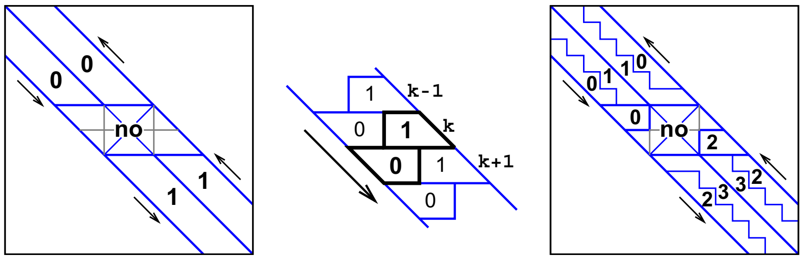


Fig. 5. Original twisted factorization method (left). Block-pipelined method: splitting of matrix L (center); combination of twisted and block-pipelined methods (right)

of the last (diagonal) element in the first row in "1". Due to this, each sweep of the Gauss procedure can be executed in two parallel threads – one for blocks "0", another for blocks "1", with the synchronization at the and of each step.

In order to implement the above scheme, it was necessary to construct the new storage scheme for all three parts of the matrix (first, last and central) and to implement proper synchronization technique for lightweight processes. Also, accurate splitting of matrices into blocks is needed in order to achieve good load balance. As a result, parallel 4-thread Gauss elimination method for a general non-symmetric sparse matrix can be developed. This combined method is illustrated on Fig. 5 (right).

This method is also "direct" with respect to the algebraic properties of the preconditioner matrix (as one described in the previous section). This is important for the solution of extremely ill-conditioned linear systems. It can be combined, for example, with 8-thread parallelization of the matrix-vector multiplication kernel in order to achieve good parallel efficiency on a two-processor system. In this case, the system's memory throughput rate will be doubled compared to a single processor, appropriately increasing performance of the solver.

Another way to increase performance is to use the single precision format (`real*4`) for the preconditioning matrix and, if possible, for the main matrix also. In this case memory access rate requirements of the algorithm will be reduced.

The considered block-pipelined method can be extended for more than two threads (for each Gauss elimination sweep). However, in this case, more accurate splitting of matrices is required that may be possible only for some sparsity patterns. When applied, this splitting will increase the parallelization potential of the kernel to 8 threads.

Parallelization efficiency results of the new method are presented in Table 1. These results were obtained for a CFD problem which matrix has 302500 unknowns and 44 non-zero elements in a half-band's row (average). Maximal half-width of the matrix is 2600. Measurements were performed on the 4-core Intel Core i7-920 processor with three DDR3-1333 memory channels.

These results demonstrate the reasonable parallelization efficiency of both methods – the twisted factorization alone, and its combination with the block-pipelined

Table 1. Parallelization results for a CFD problem

method		serial	twisted	twisted	block pipelined
threads		1	2	4	4
`real*8`	time	89.1 ms	55.6 ms	48.1 ms	41.0 ms
	speedup	1.00	1.60	1.85	2.17
`real*4`	time	89.1 ms	51.3 ms	41.6 ms	32.0 ms
	speedup	1.00	1.74	2.14	2.78

method. Another test matrix (MOSFET electronic device modeling, 77500 unknowns, 8 non-zero elements average, maximal half-width 4650) is much more sparse and less convenient for parallelization. Nevertheless, results measured for this matrix are very close (within 1-2%) to those presented in the table (with the exception of the `real*4` regime that wasn't tested).

From the table it can be seen that performance gain of using the `real*4` data format is 1.28. All these results illustrate the memory-bound property of the methods. Computational speed of the test problem on the more advanced Intel Sandy Bridge EP processor can be increased by about 1.5 times in accordance with the increase of its memory access rate. Additional speed increase can be achieved on a two-processor system with independent memory controllers.

More details on the above methods can be found in [9].

6 Conclusion

In this work we have analyzed parallelization properties and limitations of several most typical preconditioners for the Conjugate Gradient methods. This analysis confirmed that explicit preconditioners have good parallel potential and therefore remain attractive for massive parallelization. On the other hand, very efficient Modified ILU preconditioners can be moderately parallelized only for computational domains with regular geometry. In case of irregular geometry and general non-symmetric sparse matrix, only the plain ILU method can be limitedly parallelized without loosing its convergence properties. For the future development, the most promising and efficient approach is Multigrid. However, in many cases it is very difficult to implement this method. For coupled systems of equations and some other cases Multigrid can't be implemented yet due to lack of theory. For this reason, development of parallel ILU-class methods, as well as economical explicit preconditioners remains important.

Acknowledgements. This work was supported by the Russian Foundation for Basic Research (project RFBR-12-08-00034) and partly by the Institute of mathematics (IMATH) of the University of Toulon, France.

References

1. Shewchuk, J.R.: An Introduction to the Conjugate Gradient Method without the Agonizing Pain. School of Computer Science, Carnegie Mellon University, Pittsburgh (1994)
2. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS Publishing, Boston (2000)
3. 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)
4. Gustafsson, I.: A Class of First Order Factorization Methods. BIT 18, 142–156 (1978)
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. van der Vorst, H.A.: Large Tridiagonal and Block Tridiagonal Linear Systems on Vector and Parallel Computers. Par. Comp. 5, 45–54 (1987)
7. Bastian, P., Horton, G.: Parallelization of Robust Multi-Grid Methods: ILU-Factorization and Frequency Decomposition Method. SIAM J. Stat. Comput. 12, 1457–1470 (1991)
8. Fedoseyev, A., Turowski, M., Alles, M., Weller, R.: Accurate Numerical Models for Simulation of Radiation Events in Nano-Scale Semiconductor Devices. Math. and Computers in Simulation 79, 1086–1095 (2008)
9. Bessonov, O., Fedoseyev, A.: Parallelization of the Preconditioned IDR Solver for Modern Multicore Computer Systems. In: Application of Mathematics in Technical and Natural Sciences: 4th International Conference. AIP Conf. Proc., vol. 1487, pp. 314–321 (2012)

Transitive Closure of a Union of Dependence Relations for Parameterized Perfectly-Nested Loops

Włodzimierz Bielecki, Krzysztof Kraska, and Tomasz Klimek

Faculty of Computer Science and Information Technology
West Pomeranian University of Technology, ul.Żołnierska 49, 71-210 Szczecin, Poland
{wbielecki,kkraska,tklimek}@wi.zut.edu.pl

Abstract. This paper presents a new approach for computing the transitive closure of a union of relations describing all the dependences in both uniform and quasi-uniform perfectly-nested parameterized loops. This approach is based on calculating the basis of a dependence distance vectors set. The procedure has polynomial time complexity for most steps of calculations. This allows us to effectively extract both fine- and coarse-grained parallelism in loops using techniques based on applying the transitive closure of dependence relations. The effectiveness and time complexity of the approach are evaluated for loops provided by the NAS Parallel Benchmark Suite.

Keywords: transitive closure, parameterized dependence, perfectly-nested loop, parallelizing compiler.

1 Introduction

The transitive closure of a dependence relation R, R^+, makes it possible to solve reachability questions: can I reach y from x in the dependence graph represented by R? [1]. Resolving many scientific problems in such areas as databases, real-time process control, or parallel processing requires computing transitive closure. The transitive closure of dependence relations is a basic operation in many algorithms of optimizing compilers, for example, redundant synchronization removal [1], testing the legality of iteration reordering transformations [1], computing closed form expressions for induction variables [1], extracting both coarse- and fine-grained parallelism in sequential programs [2,3].

The main purpose of this paper is to present a new approach for the calculation of the transitive closure of a union of relations describing all the dependences in parameterized perfectly-nested static-control loops, where the loop bounds as well as array subscripts are symbolic parameters. Most steps of the approach are characterized by a polynomial time complexity. This approach is based on calculating the basis of a dependence distance vectors set. Having calculated transitive closure, we can apply any technique for extracting fine- or coarse-grained parallelism available in loops, for example [2,3]. The presented approach is limited to perfectly nested loops only, but it may be applied in well-known techniques for calculating the transitive closure of a union of dependence relations exposed for arbitrarily nested loops. For example, the presented technique may be used in the Floyd-Warshall algorithm [1] for calculating the transitive closure of relations representing self-dependences that is required on each iteration in this algorithm. This may lead to reducing the time complexity of the Floyd-Warshall algorithm.

2 Related Work

The work of Kelly et al. [1] introduced many of the concepts and algorithms in the computation of transitive closure. Since the transitive closure of a polyhedral relation may no longer be a polyhedral relation [1], we can, in the general case, only compute a polyhedral approximation of transitive closure.

Kelly et al. [1] consider a different set of applications which require an under-approximation of transitive closure instead of an over-approximation. Whereas Kelly et al. [1] compute an under-approximation of transitive closure, Verdoolaege [4] and Beletska et al. [5], compute an over-approximation. That is, given a relation R, they compute a relation T such that transitive closure, R^+, satisfies the condition: $R^+ \subseteq T$.

Beletska et al. [5] consider finite unions of translations, for which they compute quasi-affine transitive closure approximations, as well as some other cases of finite unions of bijective relations, which in general lead to non-affine results.

Verdoolaege [4] classifies the constraints of dependence relations to be parametric, non-parametric, and mixed. He suggests a way to calculate the transitive closure of a relation depending on a recognized relation class. He integrates many well-known techniques of calculating the transitive closure of a relation in one framework.

In paper [6], Bielecki et al. aim at exact transitive closure calculation. In the general case, this results in non-affine transitive closure.

In paper [7], Bielecki et al. apply the classical iterative least fixed point algorithm to compute transitive closure. If this process does not produce exact transitive closure after the fixed number of iterations, then the "box-closure" technique presented in paper [8] is applied. To decrease the number of iterations of the least fixed point algorithm, the authors propose to replace each simple relation with its transitive closure, provided it can be computed exactly using techniques presented in papers [6,8].

Transitive closure is also used in the analysis of counter systems to accelerate the computation of reachable sets. For example, the paper of Bardin et al. [9] and the paper of Bozga et al. [10] present ways to calculate transitive closure for this purpose.

Feautrier and Gonnord [11] and Ancourt et al. [12], focus on the computation of invariants that leads to an over-approximation of transitive closure. However, the analysis is usually performed on (non-parametric) polyhedra. Relations, for which transitive closure is computed, do not involve parameters, existentially quantified variables or unions.

All mentioned above algorithms may require much time and memory for calculating the transitive closure of a relation describing all the dependences in a loop. Experiments carried out by the authors on the NAS Parallel Benchmark Suite [13] demonstrate that such a relation may be represented as a union of hundred and even thousands simpler dependence relations. For some loops, the time of transitive closure calculation may exceeds several hundred hours. This makes impossible to apply parallelization techniques based on the transitive closure of dependence relations [2,3] for real-life code. This is why there exists a strong need in developing algorithms for calculating transitive closure characterized by reduced computational complexities in comparison with known techniques.

3 Background

In this section, we briefly introduce necessary preliminaries which are used throughout this paper.

The following concepts of linear algebra are used in the approach presented in this paper: vector, unit normal vector, vector space, field, linear combination, linear independence. Details can be found in papers [14,15,16].

Definition 1 (Column Space of a Matrix). *Let A be an $m \times n$ matrix. The space spanned by the columns of A is called the column space of A, denoted $C(A)$* [15].

Definition 2 (Basis). *A basis B of a vector space V over a field F (such as $\mathbb{R}$ or $\mathbb{Z}$) is a linearly independent subset of V that spans (or generates) V* [15]. *Every finite-dimensional vector space V has a basis* [16].

Definition 3 (Presburger Arithmetic, Presburger Formula). *We define Presburger arithmetic to be the first-order theory over atomic formulas of the form:*

$$\sum_{n}^{i=1} a_i x_i \sim c, \tag{1}$$

where a_i and c are integer constants, x_i are variables ranging over integers, and $\sim$ is an operator from $\{=, \neq, <, \leq, >, \geq\}$. The semantics of these operators are the usual ones. A formula f is either an atomic formula (1), or is constructed from formulas f_1 and f_2 recursively as follows [17]:

$$f ::= \neg f_1 | f_1 \wedge f_2 | f_1 \vee f_2.$$

In this paper, we deal with the following definitions concerned program loops: iteration vector, loop domain (index set), parameterized loops, perfectly-nested loops, details can be found in papers [18,19].

Definition 4 (Dependence). *Two statement instances $S_1(I)$ and $S_2(J)$, where I and J are the iteration vectors, are dependent if both access the same memory location and if at least one access is a write. Provided that $S_1(I)$ is executed before $S_2(J)$, $S_1(I)$ and $S_2(J)$ are called the source and destination of the dependence, respectively.*

Definition 5 (Uniform Dependence, Non-Uniform Dependence). *If the difference of iteration vectors $\boldsymbol{i_t}$ and $\boldsymbol{i_s}$ is constant for dependent statement instances $T(\boldsymbol{i_t})$ and $S(\boldsymbol{i_s})$, we call the dependence uniform, otherwise the dependence is non-uniform* [18].

Definition 6 (Dependence Distance Set, Dependence Distance Vector). *We define a dependence distance set $D_{S,T}$ as a set of differences between all such vectors T and S that stand for a pair of dependent iterations. We call each element of set $D_{S,T}$ a (dependence) distance vector and denote it as $d_{S,T}$* [18].

Definition 7 (Dependence Relation). *A dependence relation is a tuple relation of the form $\{[input\_list] \to [output\_list] : constraints\}$, where input_list and output_list are the lists of variables used to describe input and output tuples and constraints is a Presburger formula describing the constraints imposed upon input_list and output_list.*

The general form of a dependence relation is as follows [1]*:*

$$R = \{[s_i, \ldots, s_k] \to [t_i, \ldots, t_k] : \bigvee_{i=1}^{n} \exists \alpha_{i1}, \ldots\ , \alpha_{im_i}\ s.t.\ \mathcal{F}_i\},$$

where $\mathcal{F}_i, i = 1, 2, \ldots, n$ are represented by Presburger formulas, i.e., they are conjunctions of affine equalities and inequalities on the input variables $s_1, \ldots, s_k$, the output variables $t_1, \ldots, t_k$, the existentially quantified variables $\alpha_{i1}, \ldots\ , \alpha_{im_i}$, and symbolic constants.

The dependence relation is a precise description of dependences that subsumes all, even unbounded, dependence distances. Different operations on relations are permitted, such as intersection ($\cap$), union ($\cup$), difference (-), domain of relation ($domain(R)$), range of relation ($range(R)$), relation application ($R(S)$), positive transitive closure R^+, transitive closure R^*. These operations are described in detail in [20].

Definition 8 (Positive Transitive Closure). *Positive transitive closure for a given relation R, R^+, is defined as follows* [1]*:*

$$R^+ = \{e \to e' \mid\ e \to e' \in R\ \vee\ \exists e''\ s.t.\ e \to e'' \in R\ \wedge\ e'' \to e' \in R^+\}.$$

It describes which vertices e' in a dependence graph (represented by relation R) are connected with vertex e.

Definition 9 (Transitive Closure). *Transitive closure, R^*, is defined as follows* [1]*:*

$$R^* = R^+ \cup I,$$

where I is the identity relation. It describes the same connections in a dependence graph (represented by R) that R^+ does plus connections of each vertex with itself.

Definition 10 (Uniform Loop, Quasi-Uniform Loop). *We say that a parameterized loop is uniform if it induces dependences represented with the finite* *number of uniform dependence distance vectors* [18]. *A parameterized loop is quasi-uniform if all its dependence distance vectors can be represented with a linear combination of the finite number of linearly independent vectors with constant coordinates.*

Let us consider the parameterized dependence distance vector $(N, 2)$. It can be represented as the linear combination of the two linearly independent vectors $(0, 2)$ and $(1, 0)$ as follows $(0, 2) + a \times (1, 0)$, where $a \in \mathbb{Z}$.

To check whether output returned by an algorithm represents exact transitive closure, we can use the well-known fact [1] that for an acyclic relation R (for such a relation $R \cap I = \varnothing$, where I is the identity relation) the following is true:

- if R^+ is exact transitive closure, then:

$$R^+ = R \cup (R \circ R^+),$$

- if R^+ is an over-approximation, then:

$$R^+ \subset R \cup (R \circ R^+).$$

In the next section of the paper, we analyse the time complexity of the proposed approach in a machine-independent way of assessing the performance of algorithms. For this purpose, the RAM (*Random Access Machine*) model of computation is used. Under the RAM model, we measure run-time by counting up an upper bound, $\mathcal{O}$, on the number of steps an algorithm takes on a given problem instance. Details on the model and the time complexity analysis can be found in paper [21].

4 Approach to Computing Transitive Closure

The goal of the algorithm presented below is to calculate the transitive closure of a dependence relation describing all the dependences in the perfectly-nested loop. In general, such a relation is represented with a union of simpler relations [1].

4.1 Replacing the Parameterized Vector with a Linear Combination of Constant Vectors

To find constant vectors whose linear combination represents the parameterized vector, we can apply the following theorem.

Theorem 1. *Let v_p be a vector in $\mathbb{Z}^d$ and p_i, $i = 1, 2, \ldots, q$, are its parameterized coordinates, where q is the number of parameterized coordinates. We may replace vector v_p with a linear combination of constant vector v_c, $v_c \in \mathbb{Z}^d$, and unit normal vectors e_i, $e_i \in \mathbb{Z}^d$ as follows:*

$$v_p = v_c + \Sigma_i p_i \times e_i. \tag{2}$$

If $v_c = \mathbf{0}$ then v_c can be rejected from (2).

Proof. Without loss of generality, we may assume that the first n positions of v_p have constant coordinates and the last q positions have parameterized ones. Then, we can write:

$$\begin{pmatrix} c_1 \\ \vdots \\ c_n \\ p_{n+1} \\ \vdots \\ p_d \end{pmatrix} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ p_{n+1} \\ \vdots \\ p_d \end{pmatrix}, \tag{3}$$

where $d - n = q$, and further, the second vector can be written as the linear combination of unit normal vectors e_k and parameterized coefficients $p_{n+1}, \ldots, p_d$ in the last d positions:

$$\begin{pmatrix} 0 \\ \vdots \\ 0 \\ p_{n+1} \\ \vdots \\ p_d \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ p_{n+1} \\ \vdots \\ 0 \end{pmatrix} + \ldots + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ p_d \end{pmatrix} = p_{n+1} \times \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} + \ldots + \; p_d \times \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}. \tag{4}$$

Substituting (4) into (3), we obtain:

$$\begin{pmatrix} c_1 \\ \vdots \\ c_n \\ p_{n+1} \\ \vdots \\ p_d \end{pmatrix} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \\ 0 \\ \vdots \\ 0 \end{pmatrix} + p_{n+1} \times e_{n+1} + \ldots + \; p_d \times e_d, \tag{5}$$

that proves Theorem 1.

It is obvious that if $v_c = \mathbf{0}$, then v_c can be rejected without affecting the result. □

Property 1. Replacing parameterized vectors with a linear combination of vectors with constant coordinates can be done in a polynomial time.

Proof. To check each position in vector v_p, $v_p \in \mathbb{Z}^d$, the algorithm requires d operations. In the worst case, all d positions can be parameterized coordinates, hence d unit normal vectors e_k, $e_k \in \mathbb{Z}^d$ must be created. This defines $\mathcal{O}\left(d^2\right)$ time complexity of replacing parameterized vectors. □

4.2 Algorithm for Computing Transitive Closure

The idea of the algorithm presented in this section is the following. Given a set D of m dependence distance vectors in the n-dimensional integer space derived from a union of dependence relations R (it describes all the dependences in a loop), we first replace all parameterized vectors with constant vectors using Theorem 1 presented in subsection 4.1. As a result, we get k, $k \geq m$, dependence distance vectors with constant coordinates. This allows us to get rid of parameterized vectors and to form an integer matrix A, $A \in \mathbb{Z}^{n \times k}$, by inserting dependence distance vectors with constant coordinates into columns of A. The columns of A span the vector space V.

To decrease the complexity of further computations, redundant dependence distance vectors are eliminated from matrix A by finding a subset of l, $l \leq k$, linearly independent columns of A. This subset of dependence distance vectors forms the basis B, $B \in \mathbb{Z}^{n \times l}$, of A and generates the same vector space V as A does [15]. Every element of vector space V can be expressed uniquely as a finite linear combination of the basis dependence distance vectors belonging to B.

After B is completed, we can work out relation T representing the exact transitive closure R or its over-approximation. For each vertex x in the data dependence graph (where x is the source of a dependence, $x \in domain\, R$), we can identify all vertices y (the destination(s) of a dependence(s), $y \in range\, R$) that are connected with x by a path of length equal or more than 1, where y is calculated as x plus a linear combination of the basis dependence distance vectors B, i.e. $y = x + B \times z$, $z \in \mathbb{Z}^l$. The part $B \times z$ of the formula represents all possible paths in the dependence graph, represented by relation R, connecting x and y. Moreover, we have to preserve the lexicographic order for y and x, i.e. $y - x \succ 0$. Below, we present the algorithm in a formal way.

Algorithm. Calculating the exact transitive closure of a relation describing all the dependences in the parameterized perfectly-nested loop or its over-approximation.

Input: Dependence distance set $D^{n \times m} = d_1, d_2, \ldots, d_m$, where m is the number of n-dimensional dependence distance vectors.

Output: Exact transitive closure of the relation describing all the dependences in the loop or its over-approximation.

Method:

1. Replace each parameterized dependence distance vector in $D^{n \times m}$ with a linear combination of vectors with constant coordinates. For this purpose apply Theorem 1 presented in subsection 4.1.
2. Using all constant dependence vectors, form matrix A, $A \in \mathbb{Z}^{n \times k}$, $k \geq m$, whose columns span $C(A)$ [22].
3. Extract a finite subset of l, $l \leq k$, linearly independent columns from matrix $A \in \mathbb{Z}^{n \times k}$ over field $\mathbb{Z}^n$ that can represent (generate) every vector in $C(A)$. Form matrix $B^{n \times l}$, representing the basis of the dependence distance vectors set, where linearly independent vectors are represented with columns

of matrix $B^{n \times l}$. For this purpose apply the Gaussian elimination algorithm [15,16].

4. Calculate relation T representing the exact transitive closure of a dependence relation, describing all the dependences in the input loop, or its over-approximation, T, as follows:

$$T = \left\{ \begin{array}{r} [x] \to [y] \mid \exists z \; s.t. \; y = x + B^{n \times l} \times z \;\wedge\; y - x \succ 0, \; z \in \mathbb{Z}^l \;\wedge \\ \wedge\; y \in range\, R \;\wedge\; x \in domain\, R \end{array} \right\}, \quad (6)$$

where:
- R is the dependence relation describing all the dependences in the input loop,
- $B^{n \times l} \times z$ represents a linear combination of the basis dependence distance vectors d_i (the columns of $B^{n \times l}$), $1 \le i \le l$,
- $y - x \succ 0$ imposes the lexicographically forward constraints on tuples x and y of T. □

Let us demonstrate that for exact transitive closure R^+ and relation T, formed according to (6), the following condition is satisfied: $R^+ \subseteq T$. To prove this, let us note that relation T represents all possible paths between vertices x (standing for dependence sources, $x \in domain\, R$) and vertices y (standing for dependence destinations, $y \in range\, R$) in the dependence graph, represented with relation R. Indeed, a linear combination of the basis dependence distance vectors $B^{n \times l} \times z$:

- reproduces all dependence distance vectors exposed for the loop,
- describes all existing (true) paths between any pair of x and y as a linear combination of all dependence distance vectors exposed for the loop,
- can describe not existing (false) paths in the dependence graph represented by relation R.

The last case occurs when on a path between x and y, being described by T, there exists a vertex w such that $w \in range\, R \wedge w \notin domain\, R$. Such a case is presented in Figure 1, where $x_2 \in range\, R \wedge x_2 \notin domain\, R$. Relation T, built according to (6), describes the false path between x_1 and x_4 depicted with the dotted line.

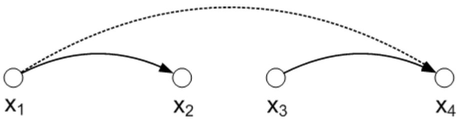


Fig. 1. False path in a dependence graph represented by relation T

Summing up, we conclude that relation T describes all existing paths in the dependence graph represented by relation R and can describe not existing paths, i.e., $R^+ \subseteq T$; when relation T does not represent false paths, $R^+ = T$.

4.3 Time Complexity

The first tree steps of the proposed algorithm can be accomplished in polynomial time.

1. As we have proved in subsection 4.1, the task of replacing parameterized vectors with a linear combination of vectors with constant coordinates can be done in $\mathcal{O}\left(d^2\right)$ operations.
2. The task of forming a dependence matrix using all k constant dependence vectors in $\mathbb{Z}^n$ requires $\mathcal{O}\left(kn\right)$ operations (memory accesses).
3. The task of identifying a set of linearly independent columns of matrix A, $A \in \mathbb{Z}^{n\times k}$ with constant coordinates to find the basis can be done in polynomial time by the Gaussian elimination algorithm. According to [23], this computation can be done in $\mathcal{O}\left(ldk\right)$ arithmetic operations.

To calculate relation T in step 4 of the algorithm, we use the Presburger arithmetic. In general, calculations based on the Presburger arithmetic are not characterized by polynomial time complexity [1].

4.4 Illustrating Example

Let us consider the following dependence distance set D:

$$D = \left\{ \begin{pmatrix}4\\0\\0\end{pmatrix} \begin{pmatrix}0\\0\\1\end{pmatrix} \begin{pmatrix}0\\1\\N1\end{pmatrix} \begin{pmatrix}2\\N2\\N3\end{pmatrix} \right\},$$

produced from the following dependence relations:

$$\begin{aligned}
R1 &= \{[i,j,k] \rightarrow [i+4,j,k] : 1 \le i \le 995 \wedge 1 \le j \le 999 \wedge 1 \le k \le 999 \quad\},\\
R2 &= \{[i,j,k] \rightarrow [i,j,k+1] : 1 \le i \le 999 \wedge 1 \le j \le 999 \wedge 1 \le k \le 998 \quad\},\\
R3 &= \left\{ \begin{array}{c} [i,j,k] \rightarrow [i,j+1,N1] : 1 \le i \le 999 \wedge 1 \le j \le 998 \wedge 1 \le N1 \le 999 \wedge \\ 1 \le k \le 999 \end{array} \right\},\\
R4 &= \left\{ \begin{array}{c} [i,j,k] \rightarrow [i+2,N2,N3] : 1 \le i \le 997 \wedge 1 \le j \le 999 \wedge 1 \le k \le 999 \wedge \\ 1 \le N2, N3 \le 999 \end{array} \right\}.
\end{aligned}$$

The presented algorithm yields the following results.

1. *Replace all parameterized dependence distance vectors.* The first parameterized vector $\begin{pmatrix}0\\1\\N1\end{pmatrix}$ is replaced with the linear combination of the vector $\begin{pmatrix}0\\1\\0\end{pmatrix}$ and the unit normal vector $\begin{pmatrix}0\\0\\1\end{pmatrix}$ as follows:

$$\begin{pmatrix}0\\1\\N1\end{pmatrix} = \begin{pmatrix}0\\1\\0\end{pmatrix} + N1 \times \begin{pmatrix}0\\0\\1\end{pmatrix}.$$

The second parameterized vector $\begin{pmatrix} 2 \\ N2 \\ N3 \end{pmatrix}$ is replaced with the linear combination of the vector $\begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$ and the two unit normal vectors $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ as follows:

$$\begin{pmatrix} 2 \\ N2 \\ N3 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix} + N2 \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + N3 \times \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

The modified dependence distance set D contains the vectors with constant coordinates only:

$$D = \left\{ \begin{pmatrix} 4 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}.$$

2. *Form a dependence matrix.* The matrix A, where all the constant dependence vectors from set D are placed in columns, is as follows:

$$A = \begin{bmatrix} 4\,0\,0\,0\,2\,0\,0 \\ 0\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,1\,0\,0\,1 \end{bmatrix}.$$

3. *Find the basis of the dependence distance set.* A set of linearly independent columns of matrix $A \in \mathbb{Z}^{n \times k}$ over field $\mathbb{Z}^n$, that can generate every vector in $C(A)$, holds the following matrix B:

$$B = \begin{bmatrix} 2\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \end{bmatrix}.$$

4. *Calculate the exact transitive closure of a dependence relation describing all the dependences in an input loop or its over-approximation, T.* Form relation T as follows:

$$T = \left\{ \begin{array}{r} [i,j,k] \rightarrow [i',j',k'] \mid \exists z\, s.t. \begin{pmatrix} i' \\ j' \\ k' \end{pmatrix} = \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{bmatrix} 2\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \end{bmatrix} \times z \,\wedge \\ \wedge \begin{pmatrix} i' \\ j' \\ k' \end{pmatrix} - \begin{pmatrix} i \\ j \\ k \end{pmatrix} \succ 0, \; z \in \mathbb{Z}^3 \,\wedge \\ \wedge \begin{pmatrix} i' \\ j' \\ k' \end{pmatrix} \in range\, R \wedge \begin{pmatrix} i \\ j \\ k \end{pmatrix} \in domain\, R \end{array} \right\} =$$

$$\left\{\begin{array}{l}[i,j,k] \rightarrow [i,j',Out\_3] : 1 \le j < j' \le 999 \wedge 1 \le i \le 999 \wedge \\ \qquad 1 \le k \le 999 \wedge 1 \le Out\_3 \le 999\end{array}\right\} \cup$$

$$\left\{\begin{array}{l}[i,j,k] \rightarrow [i,j,Out\_3] : 1 \le k < Out\_3 \le 999 \wedge 1 \le i \le 999 \wedge \\ \qquad 1 \le j \le 999\end{array}\right\} \cup$$

$$\left\{\begin{array}{l}[i,j,k] \rightarrow [i',j',Out\_3] : \exists(\alpha : 2\alpha = i + i' \wedge 1 \le i \le i' - 2 \wedge \\ \qquad 1 \le j \le 999 \wedge 1 \le k \le 999 \wedge \\ \qquad 1 \le j' \le 999 \wedge 1 \le Out\_3 \le 999 \wedge \\ \qquad i' \le 999)\end{array}\right\}.$$

Relation T represents exact transitive closure since $T = R \cup (R \circ T)$, i.e., $R^+ = T$.

5 Experiments

The goals of experiments were to evaluate the effectiveness and time complexity of the approach for loops provided by the well-known NAS Parallel Benchmark (NPB) Suite from NASA [13] and compare the results with those demonstrated with well-known techniques.

We have implemented the presented algorithm as an ANSI-C++ software module using the well-known tools: Omega Project v2.1 (including the Petit dependence analyser and OmegaCalculator) [20] and PolyLib v5.22.5 [24]. The source code of the module was compiled using the gcc compiler v4.3.0.

In order to evaluate the effectiveness and time complexity of the presented approach, we have examined loops provided by NPB, where we found 185 perfectly nested loops distributed in terms of dependence types as shown in Table 1. Only quasi-uniform loops, exposing dependences, were qualified for experiments conducted using an Intel Core2Duo T7300@2.00GHz machine with the Fedora Linux v12 32-bit operating system. Uniform loops are very simple, the transitive closure calculation takes a fraction of a second by means of each well-known approach chosen for our experiments. The results of the experiments are collected in Table 2, where time is presented in seconds.

Table 1. The quantitative distribution of perfectly-nested loops in terms of dependence types for the NAS Parallel Benchmark Suite

	Petit's dependence analysis results		Number of loops
1)	No dependences	:	123
2)	Uniform dependences, including:	:	14
	non loop-carried dependences	:	9
	loop-carried dependences	:	5
3)	Affine dependence distance vectors	:	1
4)	Parameterized dependence distance vectors	:	**47**
	The total number of perfectly nested loops	:	185

Table 2. The results of the experiments on the proposed approach to computing transitive closure (*ex*: 1 – exact result, 0 – over-approximation; Δt: difference between the transitive closure calculation time of a known correspondent technique and that of the presented approach)

#	Source loop name	Num. of relations	Proposed approach		ISL[a]		Omega[b]		Modified naive[c]		Naive iterative[d]	
			ex	t [s]	ex	Δt [s]	ex	Δt [s]	ex	Δt [s]	ex	Δt [s]
1)	BT_error.f2p_5.t	31	1	0.031426	1	0.366620	1	1.254941	1	0.352105	1	0.045889
2)	BT_rhs.f2p_1.t	46	1	0.015420	1	0.206978	1	0.266552	1	0.088564	1	0.048748
3)	BT_rhs.f2p_5.t	128	1	0.032506	1	0.628170	1	0.373698	1	0.241037	1	0.133443
4)	CG_cg.f2p_4.t	10	1	0.001394	1	0.012836	1	0.028528	1	0.033002	1	0.007343
5)	FT_auxfnct.f2p_1.t	1	1	0.000498	1	0.002367	1	0.010710	1	0.011064	1	0.005242
6)	LU_HP_jacld.f2p_1.t	2634	1	0.861241	1	3.238012	1	2.847586	1	2.850412	1	2.798487
7)	LU_HP_jacu.f2p_1.t	2364	1	0.870714	1	3.230056	1	2.775247	1	2.883089	1	2.796973
8)	LU_HP_l2norm.f2p_2.t	9	1	0.015420	1	0.071471	1	0.443246	1	0.063844	1	0.018039
9)	LU_HP_pintgr.f2p_11.t	4	1	0.003613	1	0.016903	1	0.061860	1	0.052314	1	0.151791
10)	LU_HP_pintgr.f2p_2.t	109	1	0.016306	1	0.237329	1	0.161726	1	0.104097	1	1.794602
11)	LU_HP_pintgr.f2p_3.t	6	1	0.004913	1	0.015279	1	0.086131	1	0.025622	1	0.179312
12)	LU_HP_pintgr.f2p_7.t	6	1	0.003813	1	0.014184	1	0.051845	1	0.030665	1	0.195149
13)	LU_jacld.f2p_1.t	2594	1	0.938966	1	6.642572	1	5.228918	1	5.326683	1	5.194970
14)	LU_jacu.f2p_1.t	2594	1	0.956669	1	6.579343	1	5.311930	1	5.348333	1	5.129321
15)	LU_l2norm.f2p_2.t	9	1	0.015844	1	0.072211	1	0.437792	1	0.071154	1	0.037985
16)	LU_pintgr.f2p_11.t	6	1	0.003646	1	0.014602	1	0.073625	1	0.038457	1	0.042410
17)	LU_pintgr.f2p_2.t	109	1	0.016809	1	0.259748	1	0.155823	1	0.154837	1	0.080306
18)	LU_pintgr.f2p_3.t	6	1	0.003761	1	0.040027	1	0.052203	1	0.027178	1	0.009476
19)	LU_pintgr.f2p_7.t	6	1	0.003545	1	0.014027	1	0.052915	1	0.024414	1	0.051227
20)	SP_error.f2p_5.t	31	1	0.030129	1	0.335163	1	1.124598	1	0.345867	1	0.058086
21)	SP_ninvr.f2p_1.t	103	1	0.027068	1	0.518872	1	0.360729	1	0.145716	1	0.175913
22)	SP_pinvr.f2p_1.t	103	1	0.062344	1	0.506981	1	0.311296	1	0.122795	1	2.350396
23)	SP_rhs.f2p_1.t	64	1	0.018997	1	0.280145	1	0.295704	1	0.152454	1	1.377576
24)	SP_rhs.f2p_5.t	127	1	0.037623	1	0.636079	1	0.404190	1	0.223900	1	2.894218
25)	SP_txinvr.f2p_1.t	271	1	0.060724	1	1.351747	1	0.622205	1	0.367453	1	6.027061
26)	SP_tzetar.f2p_1.t	288	1	0.065464	1	1.428076	1	0.624693	1	0.407390	1	6.309783
27)	UA_adapt.f2p_2.t	8	1	0.004565	1	0.127510	1	0.067452	1	0.074993	1	1.249933
28)	UA_diffuse.f2p_1.t	5	1	0.031599	1	0.120405	1	3.183128	1	0.082078	1	0.693391
29)	UA_diffuse.f2p_2.t	3	1	0.000841	1	0.004809	1	0.014886	1	0.015446	1	0.075129
30)	UA_diffuse.f2p_3.t	1	1	0.001557	1	0.017473	1	0.060696	1	0.065185	1	0.352796
31)	UA_diffuse.f2p_4.t	1	1	0.001319	1	0.019262	1	0.030905	1	0.029046	1	0.220720
32)	UA_diffuse.f2p_5.t	1	1	0.001318	1	0.048017	1	0.020359	1	0.073782	1	0.205120
33)	UA_precond.f2p_3.t	1	1	0.000810	1	0.008120	1	0.012220	1	0.029599	1	0.153511
34)	UA_precond.f2p_5.t	30	1	0.020345	1	0.104346	0	1.876196	1	0.236637	1	0.053637
35)	UA_setup.f2p_16.t	3	1	0.001117	1	0.012390	1	0.042547	1	0.021549	1	0.304168
36)	UA_transfer.f2p_1.t	1	1	0.001421	1	0.003180	1	0.007251	1	0.013840	1	0.088288
37)	UA_transfer.f2p_10.t	1	1	0.000563	1	0.016922	1	0.017675	1	0.010831	1	0.109149
38)	UA_transfer.f2p_13.t	1	1	0.000936	1	0.017786	1	0.061921	1	0.018809	1	0.093481
39)	UA_transfer.f2p_15.t	1	1	0.000910	1	0.008932	1	0.066882	1	0.021374	1	0.134113
40)	UA_transfer.f2p_18.t	1	1	0.001002	1	0.009408	1	0.060545	1	0.044006	1	0.092364
41)	UA_transfer.f2p_2.t	1	1	0.000567	1	0.004319	1	0.027085	1	0.009619	1	0.082240
42)	UA_transfer.f2p_3.t	1	1	0.000721	1	0.007627	1	0.015603	1	0.016645	1	0.052296
43)	UA_transfer.f2p_5.t	1	1	0.000924	1	0.008626	1	0.007632	1	0.021906	1	0.128496
44)	UA_transfer.f2p_6.t	1	1	0.000504	1	0.023024	1	0.008649	1	0.012247	1	0.057072
45)	UA_transfer.f2p_7.t	1	1	0.000800	1	0.008153	1	0.013367	1	0.015569	1	0.137118
46)	UA_transfer.f2p_8.t	1	1	0.000695	1	0.038489	1	0.008616	1	0.009248	1	0.054289
47)	UA_transfer.f2p_9.t	1	1	0.001001	1	0.058757	1	0.030424	1	0.015611	1	0.090497

[a] Integer Set Library – a library for manipulating sets and relations of integer points bounded by affine constraints (available at http://repo.or.cz/w/isl.git).

[b] Omega Project – frameworks and algorithms for the analysis and transformation of scientific programs (available at http://www.cs.umd.edu/projects/omega/).

[c] Modified Naive Iterative Method – a method described in the paper [7] and implemented by its authors in the function `IterateClosure( )` as an extension to the Omega Project (available at http://sfs.zut.edu.pl/files/omega–2.1.7.tgz).

[d] Naive Iterative Method – the well-known technique implemented in the function `compose_n( )` within the Omega Project.

Analysing the results presented in Table 2, we can conclude the approach presented in the paper to calculating transitive closure appears to be the least time-consuming in comparison with all known to the authors implemented approaches. For all loops, we obtained the shortest time of producing transitive closure. The time of transitive closure calculation by means of the presented approach is 3.2 to 275 times less than that yielded with known implementations. In all cases, the approach provides exact transitive closure.

All these facts permit us to conclude that the presented approach can be successfully applied for building parallelizing compilers permitting for automatic parallelising real-life code.

6 Conclusions

In this paper, we have presented a new approach to calculate the transitive closure of a union of relations representing dependences for both uniform- and quasi-uniform perfectly-nested parameterized loops. It is characterized with a considerably reduced time complexity in comparison with other known techniques. The algorithm has polynomial time complexity for all steps of calculations except the last one where we use the Presburger arithmetic. Carried out experiments on the NAS Parallel Benchmark Suite demonstrate that the presented technique is really fast and effective. This allows us to extract efficiently both fine- and coarse-grained parallelism in perfectly-nested loops by means of well-known techniques based on applying the transitive closure of dependence relations [2,3].

We have applied the presented algorithm in the Floyd-Warshall algorithm for calculating the transitive closure of relations representing self-dependences that is required on each iteration in this algorithm. This has permitted us to calculate the transitive closure of a union of dependence relations exposed for arbitrarily nested loops. Currently, we are carrying out experiments to evaluate the performance of such a modification of the Floyd-Warshall algorithm. The results are really promising for the time being and we are going to presents them in our next paper.

References

1. Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. In: Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1995. LNCS, vol. 1033, pp. 126–140. Springer, Heidelberg (1996)
2. Beletska, A., Bielecki, W., Cohen, A., Pałkowski, M., Siedlecki, K.: Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. Parallel Computing 37, 479–497 (2011)
3. Bielecki, W., Pałkowski, M., Klimek, T.: Free scheduling for statement instances of parameterized arbitrarily nested affine loops. Parallel Computing 38, 518–532 (2012), http://dx.doi.org/10.1016/j.parco.2012.06.001

4. Verdoolaege, S., Cohen, A., Beletska, A.: Transitive Closures of Affine Integer Tuple Relations and Their Overapproximations. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 216–232. Springer, Heidelberg (2011)
5. Beletska, A., Barthou, D., Bielecki, W., Cohen, A.: Computing the transitive closure of a union of affine integer tuple relations. In: Du, D.-Z., Hu, X., Pardalos, P.M. (eds.) COCOA 2009. LNCS, vol. 5573, pp. 98–109. Springer, Heidelberg (2009)
6. Bielecki, W., Klimek, T., Trifunovic, K.: Calculating exact transitive closure for a normalized affine integer tuple relation. Electronic Notes in Discrete Mathematics 33, 7–14 (2009)
7. Wlodzimierz, B., Tomasz, K., Marek, P., Beletska, A.: An iterative algorithm of computing the transitive closure of a union of parameterized affine integer tuple relations. In: Wu, W., Daescu, O. (eds.) COCOA 2010, Part I. LNCS, vol. 6508, pp. 104–113. Springer, Heidelberg (2010)
8. Lombardy, S., Regis-Ginas, Y., Sakarovitch, J.: Introducing VAUCANSON. Theoretical Computer Science 328(1-2), 77–96 (2004)
9. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. STTT 10(5), 401–424 (2008)
10. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 337–351. Springer, Heidelberg (2009)
11. Feautrier, P., Gonnord, L.: Accelerated invariant generation for c programs with aspic and c2fsm. Electronic Notes in Theoretical Computer Science 267, 3–13 (2010)
12. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electronic Notes in Theoretical Computer Science 267, 3–16 (2010)
13. NASA Advanced Supercomputing Division, `http://www.nas.nasa.gov`
14. Schrijver, A.: Theory of Linear and Integer Programming. Series in Discrete Mathematics (1999)
15. Shoup, V.: A Computational Introduction to Number Theory. Cambridge University Press (2005)
16. Rotman, J.: Advanced Modern Algebra, 2nd edn. Prentice Hall (2003)
17. Presburger, M.: Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. In: Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves, Warsaw, vol. 395, pp. 92–101 (1927)
18. Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Achitectures. Habilitation. Fakultät für Mathematik und Informatik Universität Passau (2004)
19. Bondhugula, U.K.R.: Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model. Dissertation. The Ohio State University (2010)
20. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega Library Interface Guide. Technical Report CS–TR–3445, Dept. of Computer Science, University of Maryland College Park (1995)
21. Skiena, S.: The Algorithm Design Manual, 2nd edn. Springer (2008)
22. Ramanujam, J.: Beyond Unimodular Transformations. Journal of Supercomputing 9, 365–389 (1995)
23. Cohen, E., Megiddo, N.: Recognizing Properties of Periodic Graphs. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 4, pp. 135–146. American Mathematical Society (1991)
24. Loechner, V.: PolyLib - A library for manipulating parameterized polyhedra. Technical report, ICPS, Université Louis Pasteur de Strasbourg (1999)

Cliff-Edge Consensus: Agreeing on the Precipice

François Taïani[1], Barry Porter[3], Geoff Coulson[4], and Michel Raynal[1,2]

[1] Université de Rennes 1, IRISA, France
{francois.taiani,michel.raynal}@irisa.fr
[2] Institut universitaire de France, France
[3] School of Computer Science, University of St Andrews, UK
bfp@st-andrews.ac.uk
[4] School of Computing and Communications, Lancaster University, UK
g.coulson@lancaster.ac.uk

Abstract. This paper presents a new form of consensus that allows nodes to agree locally on the extent of crashed regions in networks of arbitrary size. One key property of our algorithm is that it shows local complexity, i.e. its cost is independent of the size of the complete system, and only depends on the shape and extent of the crashed region to be agreed upon. In this paper, we motivate the need for such an algorithm, formally define this new consensus problem, propose a fault-tolerant solution, and prove its correctness.

Keywords: distributed computing, fault-tolerance, failure detection, consensus, scalability.

1 Introduction

Modern distributed computer systems are increasingly large and complex, often involving tens of thousands of machines distributed across continents to deliver key global services such as search, content delivery, or messaging to millions of users. Constructing such systems requires distributed services that are *scalable* to account for the global size of these systems, *efficient* to meet the increasing expectations of users, and *robust* to overcome the unavoidable failures of individual elements in such large-scale systems.

One strategy to provide these properties is to eschew any form of centralisation or global knowledge of the system, and instead rely on decentralized topologies in which each node only perceives one limited part of the system. Coordinating the work of individual nodes in such decentralized topologies is however difficult, leading to a number of works that aim to provide fundamental coordination services such as consensus in systems whose size might be unknown, and in which participants only have a partial knowledge of each other [2, 4, 8, 12, 13, 18].

In this article we look at one such fundamental service for the consistent detection of crashed regions in networks of arbitrary size. Our premise is that large-scale distributed systems can be benefit from a collective response to the crash of connected regions of the network, so that there is a need for the nodes around a crashed region to come to an agreement on the shape and extent of this region, and possibly decide on some unified recovery action to be undertaken. This problem of collective agreement can be cast as a new type of specialized consensus, where nodes that border a crashed region (i.e. nodes on the *cliff-edge*) want to agree on the extent of this crashed region (the *precipice* of our title).

This form of agreement in large-scale systems presents two interesting and related challenges, which clearly set it apart from existing works in the area: (i) The solution should be scalable, and should in particular work in networks of arbitrary size, i.e. it should only involve nodes in the vicinity of a crashed region, and never the complete system. (ii) Because of ongoing failures, nodes might disagree on the extent of a crashed region, but as they do so they will also disagree on *who* should even take part in the agreement, since both *what* is to be agreed (the crashed region), and *who* should agree on it (the nodes bordering the region) are irredeemably interdependent. We have termed this second facet of this emergent agreement the *self-constituency problem.*

Contributions: In this article, we formally define this new consensus problem, present a fault-tolerant solution that uses perfect failure detectors, and prove its correctness. Our solution works in systems of arbitrary size, in a scalable manner (the algorithm only involves nodes bordering a crashed region), for any number of faults.

Paper organization: We first present the cliff-edge consensus problem in Section 2, then move on to describe our solution (Section 3). We present our proof of correctness (Section 3.3), and finish with some related work (section 4) and a conclusion (Section 5).

2 The Problem

2.1 Overview

We consider systems in which individual nodes only have a partial knowledge of the rest of the system. This partial knowledge (Node x knows Node y) defines a form of *spatial proximity* between nodes, captured in our model by an undirected graph. (We revisit these points more formally in Section 2.2 below.) In case of correlated failures (for instance because the network's topology mirrors physical proximity as in some distributed hash table protocols [6], or because neighbouring nodes rely on the same relay to communicate), whole regions of the network might disappear, requiring surviving nodes to (i) *identify and agree* on the extent of the crashed region, and (ii) *decide* on a common action to mitigate the failure.

For instance, in the network of Fig. 1-a, the nodes in region F_1 and F_2 have crashed. These crashes are being detected by the *border nodes* (i.e. the neighbouring nodes) of each crashed region: *paris*, *london*, *madrid* and *roma* for F_1 and *tokyo*, *vancouver*, *portland*, *sydney*, and *beijing* for F_2. (This detection occurs with the help of an appropriate failure detector, which we discuss in more detail in Section 2.2.)

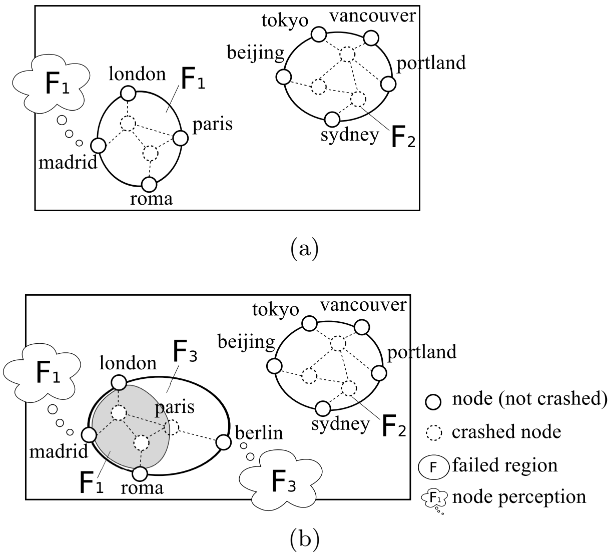


Fig. 1. Protocol instances and conflicting views

Our scalability requirement imposes that communications related to F_1 (resp. F_2) should be limited to nodes bordering F_1 (resp. F_2). For instance *vancouver* should not have to communicate with *madrid* to decide on a repair strategy for F_2. This excludes traditional consensus approaches that would involve the entire network in a protocol run.

Because of ongoing crashes, nodes bordering the same crashed region might however possess divergent views regarding the extent of their region, and hence have diverging perceptions of who should get involved in a protocol run. In Fig. 1-b, for instance, *paris* fails after *madrid* has detected F_1 as crashed, but before an agreement on F_1 has been reached. The crashed region F_1 thus grows into F_3, and a new node *berlin* (*paris*' still non-crashed neighbour) becomes involved. *berlin* detects the entirety of F_3 as crashed.

madrid and *berlin* now have different, albeit overlapping views. If *madrid* is slow to detect *paris*' crash, it might try to agree on F_1 with *london* and *roma* alone, while *berlin* will try to involve all nodes bordering F_3 to decide on F_3. Each node's effort could stall each other, or could lead to duplicated or inconsistent decisions. Our protocol prevents this and insures that any decisions pertaining to the same part of the network *converge* to a unified view, a problem that we have termed the *convergent detection of crashed regions.*

2.2 System Model and Assumptions

We model our system as a finite undirected graph $\mathcal{G} = (\Pi, \mathrm{E})$ of asynchronous message-passing nodes $\Pi = \{p_1, .., p_n\}$, where $\mathcal{G}$ represents the knowledge that nodes have of each other in the system.

A node is *faulty* if it crashes at some point, *correct* if it does not crash during the execution of the algorithm. Any two nodes might exchange messages through asynchronous, reliable, and ordered (fifo) channels. We also assume that each node can query $\mathcal{G}$ on demand, either by directly contacting live nodes, or using some underlying topology service for crashed nodes.

The *border* of a node p is the set of p's neighbours. By extension, the border of a set $S \subseteq \Pi$ of nodes are the nodes that have a neighbour in S but do not belong to S: $\mathsf{border}(S) = \{q \in \Pi \backslash S \mid \exists p \in S : (p, q) \in \mathrm{E}\}$. A *region* is a connected subgraph of $\mathcal{G}$. A *crashed region* at a time t is a region in which all nodes have crashed.

To specify the liveliness of our protocol, we need to define the three additional notions of *adjacency*, *faulty domain* and *faulty cluster*, which capture the maximum extent of crashed regions during a run. More precisely, a *faulty domain* is a region in which all nodes are faulty, but whose border nodes are correct. By construction, two faulty domains can only be either equal or disjoint.

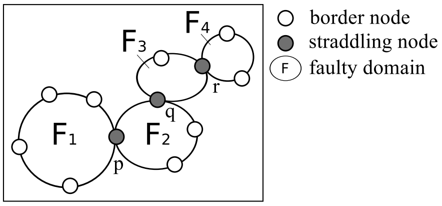


Fig. 2. A cluster of adjacent faulty domains

Two faulty domains F and H are *adjacent* (noted $F \parallel H$) if their borders intersect (e.g. $F_1 \parallel F_2$ in Fig. 2). We say that two faulty domains F_0 and F_n are in the same *faulty cluster*, noted $\mathsf{clustered}(F_0, F_n)$, if they are transitively adjacent[1], i.e. if there is a sequence of faulty domains F_i so that $F_1 \parallel F_2$ ... $F_{n-1} \parallel F_n$. For instance, we have $\mathsf{clustered}(F_1, F_4)$ in Fig. 2.

2.3 Convergent Detection of Crashed Regions: Specification

Operations. We use a mono-threaded event-based programming model to specify the convergent detection of crashed regions, and present our solution. Our service starts when a node detects one of its neighbours q as crashed ($\langle \mathbf{crash} \mid q \rangle$ event). It stops by raising a $\langle \mathbf{decide} \mid S, d \rangle$ event, where S is the crashed region decided by the local node, and d is the decision taken by this node with respect to S (e.g. a repair plan, or some other form of coordinated action). We call S the *view* of the deciding node.

[1] More formally, $\mathsf{clustered}(.,.)$ is the transitive closure of the adjacency relation, and *faulty clusters* are the equivalence classes of this closure.

Properties. The *Convergent Detection of Crashed Regions* is characterised by the following properties:

CD1 (Integrity)
No node decides twice on the same region.

CD2 (View Accuracy)
If a node p decides (V, d), then $p \in \mathsf{border}(V)$, and V is a crashed region.

CD3 (Locality)
Communication is limited to faulty-domains and their borders, i.e. a node p only exchanges messages with a node q if there is a faulty domain S such that $\{p, q\} \subseteq S \cup \mathsf{border}(S)$.

CD4 (Border Termination)
If p decides (V, d), then all correct nodes in $\mathsf{border}(V)$ eventually decide.

CD5 (Uniform Border Agreement)
If two nodes p and q decide, and p decides (V, d), and $q \in \mathsf{border}(V)$, then q decides (V, d).

CD6 (View Convergence)
If two correct nodes decide V and W, $(V \cap W \neq \emptyset) \Rightarrow (V = W)$.

CD7 (Progress)
In each faulty cluster, at least one correct node bordering a faulty domain in the cluster eventually decides: if $\mathcal{D}$ is the set of all faulty domains, $\forall V \in \mathcal{D} : \exists W \in \mathsf{clustered}(V, \cdot) : \exists p \in \mathsf{border}(W) : p$ decides.

These properties capture the requirement that the nodes bordering a crashed region should agree on the extent of this crashed region, and decide on a common course of action. CD1 (*Integrity*), CD5 (*Uniform Border Agreement*), and CD4 (*Border Termination*) are directly adapted from (uniform) consensus; CD2 (*View Accuracy*) is taken over from the strong accuracy of fault detectors; and CD7 (*Progress*) is a weak form of termination.

The problem's originality resides in the two remaining properties: CD6 (*View Convergence*) and CD3 (*Locality*). CD6 (*View Convergence*) forbids conflicting agreements on overlapping crashed regions (F_1 and F_3 in Fig. 1). CD3 (*Locality*) provides scalability by limiting the system's reaction to the vicinity of crashed regions. In particular, CD3 (*Locality*) implies that nodes with no faulty neighbours do not take part in the protocol. As a result, the protocol only depends on the amount of failures in the system, but not on the system's actual size. *Locality* also excludes the use of a system-wide consensus to fulfil the other properties.

This *Locality* property creates however a pernicious inter-dependency between the protocol's participants (the 'constituency') and what they are agreeing to: To start our protocol, a node needs to know with whom it should be agreeing (its fellow border nodes), but this set of nodes depends on the final outcome of the protocol (the crashed region agreed upon).

In the following, we present a solution to this problem and propose a proof of its correctness.

3 A Cliff-Edge Consensus Protocol

3.1 Preliminaries: Failure Detector, Multicast, Region Ranking

Our algorithm uses a perfect failure detector provided in the form of a *subscription-based* service: a node p subscribes to the crashes of a subset of nodes S by issuing the event $\langle$**monitorCrash** $| S\rangle$ to its local failure detector. Our failure detector is perfect and ensures: *(i) Strong Accuracy*: if a node p receives a $\langle$**crash** $| q\rangle$ event, then q has crashed, and p did subscribe to be notified of q's crash; and *(ii) Strong Completeness*: if a node q has crashed, and p has subscribed to be notified of q's crash, then p will eventually receive a $\langle$**crash** $| q\rangle$ event.

For the sake of conciseness, we use a basic multicast service, represented by the events $\langle$**multicast** $| R$, [m]$\rangle$ and $\langle$**mDeliver** $| p$, [m]$\rangle$. This service simply sends to each recipient a multicast message [m] over underlying point-to-point channels, in a plain loop. This service provides no guarantees beyond those of the underlying channels, and is essentially a shorthand to keep our code brief.

We also use a *ranking relation* between regions, noted $\succ$: $R \succ S$ iff either (i) R contains more nodes than S, or (ii) they contain the same number of nodes but R's border contains more nodes than S's border, or (iii) R and S have the same size, and so do their respective borders, but R is greater than S according to some strict total order relation $\rhd$ on sets of nodes. The actual ordering relation $\rhd$ on node sets does not matter. One possibility is to use a lexicographic order on node IDs. By construction, $\succ$ is a strict total order on regions. For a set $\mathcal{C}$ of regions, $\mathsf{maxRankedRegion}(\mathcal{C})$ is the highest ranked region in $\mathcal{C}$.

Finally, for a subset S of nodes, $\mathsf{connectedComponents}(S)$ returns the set of the maximal regions of S, i.e., formally, the vertex sets of the connected components of the subgraph $\mathcal{G}[S]$ induced by S in $\mathcal{G}$. (Remember that a region is defined as a connected subgraph of $\mathcal{G}$, see Section 2.2.)

3.2 Algorithm

The pseudo code of our algorithm is given in Figure 1. $\langle$**init**$\rangle$ is executed by all nodes when the protocol starts. Each node then remains idle until one of its neighbours fails, as notified by a $\langle$**crash** $| q\rangle$ event.

The bulk of the protocol is primarily a superposition of flooding uniform consensus instances [9, 14] between the border nodes of proposed views. This superposition is complemented by *an arbitrating mechanism* to deal with overlapping but conflicting views (line 26). Because of this arbitration, all consensus instances must be tracked concurrently by our protocol, in the variables opinions$[\cdot][\cdot][\cdot]$ and waiting$[\cdot][\cdot]$, which are indexed by proposed views (in addition to rounds, and, for opinions, participants).

A node starts a consensus instance when it detects that one of its neighbours has crashed (line 17). The view it proposes has been incrementally built when receiving $\langle$**crash** $| .\rangle$ events (line 5), and is the highest ranked crashed region known to the node at this point. The view construction continues in the background as the consensus unfolds (lines 5-10), to be used if the attempt to reach an agreement fails.

The opinion vectors received from other nodes in a round are gathered at line 18. Because a node might be involved simultaneously in multiple conflicting consensus instances, messages related to conflicting views are also gathered and processed. The resulting opinion vectors, indexed by round and proposed view (line 24) are stored in opinions$[\cdot][\cdot][\cdot]$.

If a node becomes aware of a conflicting view with a lower rank (line 26), it sends a special $\text{op}_{\mathsf{reject}}$ vector to this view's border nodes, and subsequently ignores any message related to this view (lines 28-31).

Rounds are completed at line 32 when all non-crashed border nodes of view have replied: if no more rounds are needed (line 34), and the node's final vector only contains accept values, a decision value is deterministically selected for the proposed view (line 35), and the node decides[2]. Otherwise the whole process is reset, and restarts at line 12 as soon as a new crashed node is detected.

3.3 Proof of Correctness

In the following, we use a subscript notation to distinguish between the same protocol variable at different nodes: e.g. opinions_p for the variable opinions of p.

Theorem 1. *Our protocol fulfils properties CD1 (Integrity), CD2 (View Accuracy), and CD3 (Locality).*

Proof. CD1 is fulfilled by construction. For CD2, connectedComponents() at line 8 and the strong accuracy of the failure detector insure that proposed views are crashed regions. Using recursion on $\langle \textbf{crash} \mid . \rangle$ events, a node p can be shown to respect the two invariants *(i)* $p \in \mathsf{border}(\text{locallyCrashed}_p)$ and *(ii)* $\{p\} \cup \text{locallyCrashed}_p$ is connected, thus yielding that p is on the border of any view it proposes. CD3 follows from CD2, and the fact that two nodes only exchange messages when both border a region detected as failed by one of them.

Our proof of the remaining four properties reuses elements of the proof of the consensus algorithm presented in [9] for strong failure detectors (S), of which the flooding uniform consensus is derived. The difficulty lies in that our protocol uses multiple overlapping consensus instances, each indexed by the view it proposes, with no prior agreement on either the set the consensus instances, their participants, or their sequence. In addition, our arbitrating mechanism means a node can first propose and then reject the same view, thus complicating the *uniform border agreement*, as we shall see.

Lemma 1. *At any execution point the vectors* $\text{opinions}_p[V][r][\cdot]$ *of* p *are such that* $\forall q \in \mathsf{border}(V)$:
1) $\text{opinions}_p[V][r][q] = \mathsf{reject} \Rightarrow q$ *rejected* V *earlier* $\wedge$
2) $\text{opinions}_p[V][r][q] = (\mathsf{accept}, \cdot) \Rightarrow q$ *accepted* V *earlier*

[2] For clarity's sake, the presented version is not optimized. A classical optimization consists in terminating a consensus instance once a node sees that all nodes in its border set know everything (i.e. no $\perp$), i.e. after two rounds, in the best case.

Algorithm 1. Convergent detection of crashed regions executed by node p

```
1: upon event ⟨init⟩
2:     decided ← ⊥ ; proposed ← ⊥
3:     locallyCrashed, maxView, candidateView, $V_p$, received, rejected ← ∅
4:     trigger ⟨monitorCrash | border($p$)⟩
5: upon event ⟨crash | $q$⟩                                          ▷ View construction
6:     locallyCrashed ← locallyCrashed ∪ {$q$}
7:     trigger ⟨monitorCrash | border($q$)\locallyCrashed⟩
8:     $\mathcal{C}$ ← connectedComponents(locallyCrashed)
9:     if maxView ≺ maxRankedRegion($\mathcal{C}$) then
10:        maxView ← maxRankedRegion($\mathcal{C}$)
11:        candidateView ← maxView
12: upon event proposed = ⊥ ∧ candidateView ≠ ∅                  ▷ New consensus instance
13:     $V_p$ ← candidateView ; candidateView ← ∅
14:     proposed ← selectValueForView($V_p$)
15:     $\mathrm{op}_{\mathsf{accept}}[p_k]$ ← ⊥  for all $p_k \in$ border($V_p$)\{$p$}
16:     $\mathrm{op}_{\mathsf{accept}}[p]$ ← (accept, proposed) ; $r$ ← 1
17:     trigger ⟨multicast | border($V_p$), [1, $V_p$, border($V_p$), $\mathrm{op}_{\mathsf{accept}}$]⟩        ▷ proposing $V_p$
18: upon event ⟨mDeliver | $p_i$,[$r, V, B$, op]⟩ ∧$V \notin$ rejected          ▷ Updating opinions
19:     if $V \notin$ received then
20:        received ← received ∪ {$V$}                        ▷ Initialise data structures for $V$
21:        opinions[$V$][$r$][$p_k$] ← ⊥ for all $p_k \in B \wedge 1 \leq r < |B|$
22:        waiting[$V$][$r$] ← $B$ for all $1 \leq r < |B|$
23:     for all $p_k$ such that (opinions[$V$][$r$][$p_k$] = ⊥ ∧ op[$p_k$] ≠ ⊥) do
24:        opinions[$V$][$r$][$p_k$] ← op[$p_k$]
25:     waiting[$V$][$r$] ← waiting[$V$][$r$]\ ({$p_i$} ∪ {$p_k$|op[$p_k$] = reject})
26: upon event ∃$L$ ∈ received : $L \prec V_p$                     ▷ Rejecting a lower ranked view
27:     trigger ⟨reject | $L$⟩
28: upon event ⟨reject | $L$⟩
29:     $\mathrm{op}_{\mathsf{reject}}[p_k]$ ← ⊥  for all $p_k \in$ border($L$)\{$p$}
30:     $\mathrm{op}_{\mathsf{reject}}[p]$ ← reject; received ← received\{$L$}; rejected ← rejected ∪ {$L$}
31:     trigger ⟨multicast | border($L$), [1, $L$, border($L$), $\mathrm{op}_{\mathsf{reject}}$]⟩
32: upon event $V_p$ ∈ received ∧ waiting[$V_p$][$r$]\locallyCrashed = ∅ ∧ decided = ⊥
33:     if $r \geq$ |border($V_p$)| − 1 then                          ▷ Consensus instance completed
34:        if ∀$p_i$ ∈ border($V_p$) : opinions[$V_p$][$r$][$p_i$] = (accept, $v_{p_i}$) then
35:           decided ← deterministicPick($\{v_{p_i}\}_{p_i \in \mathsf{border}(V_p)}$)             ▷ Decision
36:           trigger ⟨decide | $V_p$, decided⟩
37:        else proposed ← ⊥                                  ▷ Consensus attempt failed, reset
38:     else                                                              ▷ New round
39:        $r \leftarrow r + 1$
40:        trigger ⟨multicast | border($V_p$), [$r$, $V_p$, border($V_p$), opinions[$V_p$][$r-1$]]⟩
```

Proof. First let us note that the only location where opinions[V][q][q] is explicitly assigned an accept (resp. reject) value is when q accepts (resp. rejects) view V at line 16 (resp. line 30). This accept (resp. reject) value then propagates to the opinion vectors of other border nodes through the network (lines 17, 31 and 40), and the assignment of line 24. A recursive data-flow argument on the values of opinions[V][r][$\cdot$] taken at these lines yields the lemma.

Lemma 2. *A node proposes (resp. rejects) a given view V at most once. A node never proposes a view it has previously rejected.*

Proof. The uniqueness of rejection follows from the use of the rejected and received variables. The use of the strict ranking relation $\prec$ (line 9) means the series of values taken by the variable candidateView is strictly monotonic according to $\prec$ (line 11), and that by construction this is also true of V_p (line 13), thus completing the lemma.

Lemma 3. *If two nodes p and q complete a consensus instance on the same view* $\mathrm{V}_{p|q} = V$ *(line 34), they obtain the same opinion vector:*

$$\text{opinions}_p[V][N][\cdot] = \text{opinions}_q[V][N][\cdot]$$
$$\textit{where } N = |\mathsf{border}(V)| - 1$$

Proof. We prove this lemma by contradiction. Let's assume $\exists k \in \mathsf{border}(V)$: $\text{opinions}_p[V][N][k] \neq \text{opinions}_q[V][N][k]$. If one of the two values is $\bot$, we can use the well-known argument on cascading crashes, identifying N distinct nodes in $\mathsf{border}(V)$ that did not complete the consensus instance, contradicting the fact that p and q completed it.

Let's now assume both values are non-$\bot$. The first sub-case is when both values are accept for k, with different decision values on p and q, i.e. $\text{opinions}_p[V][N][k] = (\mathsf{accept}, v_k^p)$ and $\text{opinions}_q[V][N][k] = (\mathsf{accept}, v_k^q)$ with $v_k^p \neq v_k^q$. Using lemma 2, we conclude that line 16 is executed only once by k for V, and that $v_k^p = v_k^q$, yielding the contradiction.

Finally, let's assume one value is accept, while another is reject, e.g. without loss of generality, $\text{opinions}_p[V][N][k] = (\mathsf{accept}, \cdot)$ and $\text{opinions}_q[V][N][k] = \mathsf{reject}$. From lemma 1 we conclude that k has both proposed and rejected V. Let's call e^k_{accept} and e^k_{reject} the corresponding execution points. Because of lemma 2, e^k_{accept} and e^k_{reject} are unique, and e^k_{accept} happened before e^k_{reject}. Because the best-effort multicast is fifo, this means q received the message for e^k_{accept} before that of e^k_{reject}, and because line 24 only updates $\bot$ values, that $\text{opinions}_q[V][N][k] = (\mathsf{accept}, \cdot)$, yielding the contradiction.

Theorem 2. *Our protocol fulfils properties CD5 (Uniform border agreement) and CD4 (Border termination).*

Proof. Let's assume p and q decide, p decides $(V_p, \text{decided}_p)$, and $q \in \mathsf{border}(V_p)$. If p decides on V_p, then p completed the corresponding consensus instance with only accept values, and since $q \in \mathsf{border}(V_p)$ we have $\text{opinions}_p[V_p][N][q] = (\mathsf{accept}, \cdot)$. By lemma 1, q proposed V_p.

Let us now note that by construction a node cannot propose any new view once it has decided on one (Remark **N1**). That is because of the guards decided $= \perp$ at line 32 and proposed $= \perp$ at line 12. The first guard means the lines 32-40 are never executed again once a value has been decided at lines 35-36. In particular the variable proposed is never reset at line 37 after a decision. This in turn means a node cannot propose a new view at lines 12-17.

Similarly, the same guard proposed $= \perp$ at line 12 means a node cannot start a new consensus instance before completing the current one (Remark **N2**). Remarks **N1** and **N2** thus mean q completed the consensus instance corresponding to V_p before deciding. By lemma 3, q obtained the same vector opinions$_q$ as p on V_p, and hence decided $(V_p, \text{decided}_p)$ by determinism of $\mathsf{deterministicPick}$ (line 35), thus proving CD5.

CD4 follows the same reasoning, with the observation that if a node p completes a consensus instance on a view V_p, then all other nodes in $\mathsf{border}(V_p)$ either took part in each round or crashed, implying that all correct nodes eventually complete the instance with the same opinion vector as p (by way of lemma 3).

Theorem 3. *Our protocol fulfils CD6 (View convergence).*

Proof. Let's consider two correct nodes p and q that decide on overlapping crashed regions V_p and V_q: $V_p \cap V_q \neq \emptyset$. If one node is in the border of the other's region, e.g. $p \in \mathsf{border}(V_q)$, then *Uniform Border Agreement* (CD5) and *Integrity* (CD1) give us $V_p = V_q$.

Let's now assume $p \notin \mathsf{border}(V_q) \wedge q \notin \mathsf{border}(V_p)$, and use a proof by contradiction. Since $V_p \cap V_q \neq \emptyset$, there is a node $a \in V_p \cap V_q$ (Fig. 3). V_p being a region bordered by p (CD2), there exists a path $(n_0 = p, n_1, ..., n_k = a)$ that links a to p through V_p: $\{n_1, ..., n_k, a\} \subseteq V_p$. Since $a \in V_q$, we can consider the point when this path "penetrates" for the first time into V_q, i.e. we can consider $n_{i_0} \in V_q$ and $\forall i < i_0 : n_i \notin V_q$. Since p is correct, $n_{i_0} \neq p$, i.e. $i_0 \geq 1$, and we can look at n_{i_0-1}, the node in the path just before n_{i_0}. Let's call this node r (Fig. 3). Because n_{i_0} is the first node in the path to belong to V_q, we have $r \in \mathsf{border}(V_q)$, and since $p \notin \mathsf{border}(V_q)$, $r = n_{i_0-1}$ cannot be p ($i_0 > 1$). Because, with the exception of p, the path connecting p to a is embedded in V_p, this means that r is in fact located in p's crashed region. This reasoning thus yields us a node (r) that is both on $\mathsf{border}(V_q)$ *and* in p's crashed region: $r \in V_p \cap \mathsf{border}(V_q)$. Using an identical argument, we can find a node s such that $s \in V_q \cap \mathsf{border}(V_p)$ (Fig. 3).

To complete our proof, we now look at the happen-before relationships between events related to r and s. Let's first consider s. Since $s \in \mathsf{border}(V_p)$ and p decided on V_p, s itself did propose V_p (lemma 1). Since $r \in V_p$, s did detect r as crashed as some point. By a similar reasoning, we conclude that r proposed V_q, and hence detected s as crashed as some point.

We thus end up with a set of 6 events that form a circular chain of happen-before events: *s_detects_r* $\rightarrow$ *s_proposes* $\rightarrow$ *s_crashes* $\rightarrow$ *r_detects_s* $\rightarrow$ *r_proposes* $\rightarrow$ *r_crashes* $\rightarrow$ *s_detects_r* ... This provides our contradiction.

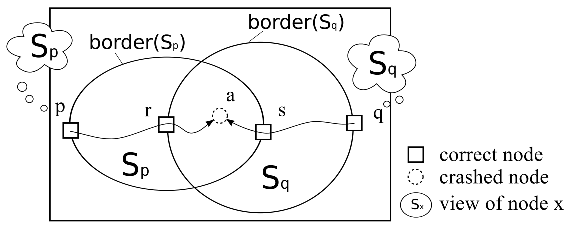


Fig. 3. Convergence between overlapping views

Theorem 4. *Our protocol fulfils properties CD7 (Progress).*

Proof. Again we use a contradiction: consider a cluster of adjacent faulty domains (Fig. 2), and assume none of its correct border nodes ever decide. Since this situation lasts indefinitely, we can consider the case where all crashed regions are maximal and all remaining nodes are correct.

Because the views proposed by a node are strictly monotonic according to $\prec$, and because $\mathcal{G}$ is finite, a node cannot propose an infinite sequence of views. A correct border node p that does not decide falls therefore into two cases: either (**C1**) p is blocked waiting for the reply of another node q (line 38); or (**C2**) the last view proposed by p failed (line 37), and p does not detect any new crashed node (line 5).

Case C1 : If p is waiting for the reply of some other node q, q must be correct (if it were not, q would eventually crash, thus unblocking p). Since there is a path of crashed nodes from p to q (since p is waiting for q), q is on the border of the same faulty domain as p, so q never decides (by assumption).

As for p, q falls in either case **C1** or **C2**. Let's first assume that the last view V_q^{max} proposed by q failed, and q does not detect any new crashed node (**C2**). Since we have assumed that all faulty nodes have crashed, by strong completeness of the failure detector, V_q^{max} is a faulty domain, and because of the use of maxRankedRegion (line 10) and the fact that $\prec$ subsumes set inclusion, V_q^{max} is higher ranked than any crashed region bordered by q.

Since p is waiting for q, $V_p \neq V_q^{\mathsf{max}}$, and since q is on the border of both V_p and V_q^{max}, V_p is lower-ranked than V_q^{max}: $V_p \prec V_q^{\mathsf{max}}$. q has received a round-1 message proposing V_p (line 18), and should have rejected it (line 31), thus ending p's wait on q, which contradicts our assumption.

We therefore conclude that q cannot fall in case **C2**, and instead is blocked in a consensus round proposing a crashed region V_q (case **C1**). q received p's proposal message, and did consider it for rejection (line 26). Because p is waiting for q, we know it did not receive any rejection message from q, and therefore, $V_p \succeq V_q$. Since p is waiting for q, q is not proposing the same view as p, yielding a strict ordering between the two views $V_p \succ V_q$.

This construction can be repeated recursively, first for q, and then for the node q is waiting on, etc, each time yielding an infinite number of pairwise distinct crashed regions (via CD2) that are strictly ordered by the ranking relationship:

$V_{p_1} \succ V_{p_2} \succ ... \succ V_{p_i} \succ ...$ This contradicts our assumption that each faulty cluster contains a finite number of faulty domains, each containing a finite number of nodes.

Case C2 : Let's now assume the last view V_p^{max} proposed by p failed, and p does not detect any new crashed node. As with V_q^{max} above, V_p^{max} is a faulty domain, and all its border nodes are correct. Because the failure detector is strongly accurate, for p's proposal to fail, one node $q \in \mathsf{border}(V_p^{\mathsf{max}})$ must have rejected V_p^{max} because it was proposing a higher-ranked view V_q^{higher}. By assumption, q never decides, it must either fall in case **C1** or **C2**. If q is in case **C1**, we can repeat the same argument as for p in Case **C1**, above. If q is in case **C2**, q's last view V_q^{max} is higher or equal than any view q ever proposed, implying $V_p^{\mathsf{max}} \prec V_q^{\mathsf{higher}} \preceq V_q^{\mathsf{max}}$.

By recursively applying this argument, we either come back to case **C1** at some point, or obtain an infinite sequence of strictly ordered faulty domains $V_{p_1}^{\mathsf{max}} \prec V_{p_2}^{\mathsf{max}} \prec V_{p_3}^{\mathsf{max}} \prec ...$, which yields our contradiction.

4 Related Work

The algorithm we presented builds on our earlier work on the generic repair of overlay networks [17], in which we first sketched some of the ideas presented in this paper, albeit without any formal definition or proof.

Our algorithm can be viewed as a combination of an ad-hoc group formation and 'preference-based' leader election [19], with the important difference that the algorithm attempts to find a *stable* region of a network (crashed region) to operate on.

Consensus [5, 9] and leader election [15, 19] are both well-studied fields, although most approaches do not address the ad-hoc group formation problem; i.e. the inter-dependency that arises between those who are *agreeing* (the border set) and that which they are *agreeing to* (the crashed region, and thus constituency of the border set itself). Our work has however some similarities with consensus with unknown participants, where the set of participants is fixed, but unknown to the nodes involved [3, 7, 8, 13]. These works introduce the notion of a participant detector (PD) and study the properties this detector should fulfil to permit consensus under different assumptions.

These works are however quite different from what we are proposing, in that in our case participants are not only unknown, but evolve as failures occur. Our work also puts a strong focus on scalability with the *locality* property.

The service we propose is also related to group membership [10]. Deciding on a view in our protocol can be seen as the equivalent of installing a view. The link is particularly true with partitionable group membership (PGM) services [1, 11, 16], which look at how successively installed views should evolve to ensure that both reachability and unreachability between nodes are reflected in their installed views.

As in partitionable group membership, our service requires views held by nodes to converge when these nodes enter a particular relationship. This relationship depends on reachability in PGM, while ours arise when two nodes propose views that overlap (CD6).

The key difference however is that, whereas PGM services are defined in terms of eventual convergence of installed views, our specification is stricter in that nodes can only decide once on a given region (CD1), and must therefore detect when they have reached a convergent state, while insuring liveliness in the system (CD7).

5 Conclusion

In this paper we have formally specified a service for the convergent detection of crashed regions, where the nodes of an arbitrary large distributed system attempt to reconcile their views of neighbouring crashed regions. We have described a fault-tolerant solution to this problem, and proved its correctness. One key aspect of our specification is that it only involves nodes bordering a crashed region (*locality*), and requires nodes to explicit decide when they have converged on a unified view.

Beyond the detection of correlated crashed regions, we think this form of agreement can be seen as a particular case of a wider class of algorithms that attempt to create local collective knowledge about some distributed condition in a manner that is both deterministic and scalable. Scalability here means costs only depend on the 'extent' of the knowledge to be constructed, independently of the actual size of the system, a powerful property in very large systems.

Being crashed can also be seen as a particular case of stable property. It could be interesting to see how this work could be extended to the detection of connected regions of nodes that share a given stable predicate (say a particular stable state). A further challenge could be to investigate how the notion of predicate-based regions and the properties of the corresponding agreement protocols could be evolved to tackle unstable properties.

References

1. Babaoglu, O., Davoli, R., Montresor, A.: Group communication in partitionable systems: Specification and algorithms. Tech. Rep. UBLCS-98-01, University of Bologna (1998)
2. Baldoni, R., Bertier, M., Raynal, M., Tucci-Piergiovanni, S.: Looking for a definition of dynamic distributed systems. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 1–14. Springer, Heidelberg (2007)
3. Bar-Joseph, Z., Keidar, I., Lynch, N.: Early-delivery dynamic atomic broadcast. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 1–16. Springer, Heidelberg (2002)
4. Bonnet, F., Raynal, M.: The price of anonymity: Optimal consensus despite asynchrony, crash, and anonymity. ACM Trans. Auton. Adapt. Syst. 6(4), 23:1–23:28 (2011)

5. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM 32(4), 824–840 (1985)
6. Castro, M., Druschel, P., Hu, Y.C., Rowstron, A.: Exploiting network proximity in distributed hash tables. In: International Workshop on Future Directions in Distributed Computing (FuDiCo), pp. 52–55 (June 2002)
7. Cavin, D., Sasson, Y., Schiper, A.: Reaching agreement with unknown participants in mobile self-organized networks in spite of process crashes. Research Report IC/2005/026, EPFL (2005)
8. Cavin, D., Sasson, Y., Schiper, A.: Consensus with unknown participants or fundamental self-organization. In: Nikolaidis, I., Barbeau, M., An, H.-C. (eds.) ADHOC-NOW 2004. LNCS, vol. 3158, pp. 135–148. Springer, Heidelberg (2004)
9. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. of the ACM 43(2), 225–267 (1996)
10. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comp. Surveys 33(4), 427–469 (2001)
11. Fekete, A., Lynch, N., Shvartsman, A.: Specifying and using a partitionable group communication service. ACM Trans. Comput. Syst. 19(2), 171–216 (2001)
12. Friedman, R., Raynal, M., Travers, C.: Two abstractions for implementing atomic objects in dynamic systems. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 73–87. Springer, Heidelberg (2006)
13. Greve, F., Tixeuil, S.: Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: Proc. of the 37th IEEE/IFIP Int. Conf. on Dependable Sys. and Networks, DSN 2007, pp. 82–91 (2007)
14. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming, 300 p. Springer (2006) ISBN 978-3540288459
15. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Inf. and Comp. 88(1), 60–87 (1990)
16. Keidar, I., Sussman, J., Marzullo, K., Dolev, D.: Moshe: A group membership service for WANs. ACM Trans. Comput. Syst. 20(3), 191–238 (2002)
17. Porter, B., Taïani, F., Coulson, G.: Generalised repair for overlay networks. In: 25th Proc. of the IEEE Symp. on Reliable Dist. Syst., SRDS 2006, pp. 132–142 (2006)
18. Raynal, M.: Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. Synthesis Lectures on Distributed Computing, 273 p. Morgan & Claypool (2010) ISBN 978-1-60845-293-4
19. Singh, S., Kurose, J.F.: Electing “good” leaders. J. of Parallel and Distributed Comp. 21, 184–201 (1994)

Hybrid Multi-GPU Solver Based on Schur Complement Method

Sergey Kopysov, Igor Kuzmin, Nikita Nedozhogin, Alexander Novikov, and Yulia Sagdeeva

Institute of Mechanics UB RAS,
34 T. Baramzinoy, Izhevsk, Russia 426067
{s.kopysov,imkuzmin}@gmail.com, nedozhogin@inbox.ru,
sc_work@mail.ru, sagdeeva@yandex.ru
http://udman.ru/iam

Abstract. This paper presents a new hybrid solver based on Schur complement method, in which computations are distributed between multiple CPUs and GPUs. In this solver, the Schur complement is computed either on CPUs (for small problem size) or on GPUs (for large problem sizes). To solve the interface system, we propose a new multi-GPU algorithm that implements conjugate gradient method with explicit preconditioning. Experiments with wrap spring simulation on hybrid multi-core multi-GPU cluster demonstrate efficiency of the proposed method.

Keywords: hybrid solver, domain decomposition, Schur complement, sparse matrices, hybrid CPU/GPU platforms, CUDA.

1 Introduction

Hybrid solvers for linear systems [1] combine the advantages of both direct and iterative methods. In hybrid solvers, usually, a direct method is used for factorization of matrix blocks, and then an iterative method is used to compute the Schur complement. The most widely used parallel software tools for hybrid methods are HIPS (Hierarchical Iterative Parallel Solver) [2], MaPHyS (Massively Parallel Hybrid Solver) [3], PDSLin (Parallel Domain decomposition Schur complement based Linear solver) [4] and ShyLU (Scalable hybrid LU) [5]. Each of these tools provides a variant of the Schur complement method, such as multifrontal method, Krylov subspace methods with preconditioning (conjugate gradient method, generalized minimum residual method). In some cases, the Schur complement method is used for preconditioning. High performance algorithms for these methods are implemented for distributed- and shared-memory platforms, including multi-cores.

Computing potential of hybrid solvers can be revealed on hybrid CPU/GPU platforms. Programming systems for general-purpose computing on GPU, such as CUDA, OpenCL and OpenACC, accelerate linear algebra routines ten to hundred times in comparison to CPU. Thousands of parallel GPU threads can efficiently perform a large number of simple arithmetic operations, such as multiplication and addition of the elements of matrices and vectors. However, sequential operations and branching, which are typical for direct methods (matrix factorization into triangular multipliers), are performed on GPU slower than on CPU cores. Parallel computing on multiple GPUs involves the following programming techniques: CUDA + OpenMP for multiple GPUs within a computing module, and CUDA + MPI for multiple GPU-accelerated modules connected by network.

Efficient utilization of GPUs in the Schur complement method depends on optimal distribution of computation between CPUs and GPUs and optimal decomposition of matrices. In this paper, the Schur complement method is designed as a hybrid numerical method for hybrid platforms. It is applied to the linear systems of equations obtained from 3D elasticity problems. The partition of the finite-element mesh is used to form the matrix blocks of the system.

The paper is structured as follows. In Section 2, we briefly introduce the Schur complement method. In Section 3, we analyze the matrix storage formats for this method in terms of time and space complexity. In Section 4, we present a new parallel hybrid solver based on the Schur complement method. Experiments with a wrap simulation application demonstrate the efficiency of our solution on multi-GPU platforms.

2 The Schur Complement Method

We consider the Schur complement method as a version of domain decomposition method, also called substructuring method [6]. To ensure the independence of the calculations in the individual subdomains and the subsequent interaction between the subdomains, all nodes of the computational mesh that approximates the domain are split into two subsets: interior and boundary nodes.

Let the domain Ω be split into n_Ω non-overlapping subdomains

$$\Omega = \Omega_1 \bigcup \Omega_2 \bigcup \ldots \bigcup \Omega_{n_\Omega}, \quad \text{where} \quad \Omega_i \bigcap \Omega_j = \emptyset, \quad \Gamma_B = \bigcup_{i=1}^{n_\Omega} \partial\Omega_i \setminus \partial\Omega. \tag{1}$$

This partition into subdomains results from the partition of the dual graph of the computational mesh $G(V,E) = \bigcup_{i=1}^{n_\Omega} G_i(V_i,E_i)$, where:

- the set of the graph nodes V is a set of the finite elements of the mesh,
- the set of the graph edges E is a set of the adjacent finite elements,
- $V_i \subset V$ is a set of the finite elements of a subdomain.

We assume all the subgraphs $G_i(V_i,E_i)$ are connected, otherwise the system of equations (2) for the subdomain Ω_i can be split into disconnected systems of equations.

The set of mesh nodes $\hat{V}$ consists of interior nodes $\hat{V}_{I_i}$, which belong to the interior of a subdomain associated with the subgraph $G_i(V_i,E_i)$, and boundary nodes $\hat{V}_{B_i}$, which belong to the subdomain boundary. The set of boundary nodes includes interface nodes $\hat{V}_{C_i} \subset \hat{V}_{B_i}$ that are shared by neigboring subdomains.

For each subdomain Ω_i, a system of equations is constructed in such a way that the unknowns corresponding to the interior and boundary nodes are separated:

$$\begin{pmatrix} A^i_{II} & A^i_{IB} \\ A^i_{BI} & A^i_{BB} \end{pmatrix} \begin{pmatrix} u^i_I \\ u^i_B \end{pmatrix} = \begin{pmatrix} f^i_I \\ f^i_B \end{pmatrix}, \tag{2}$$

where subscripts I, B stand for interior and boundary degrees of freedom.

The system for the interface nodes is defined as follows:

$$S_{BB}\tilde{u}_B = \tilde{f}_B, \tag{3}$$

where $S_{BB} = \sum_i^{n_\Omega} \left(A^i_{BB} - A^i_{BI} A^{i^{-1}}_{II} A^i_{IB} \right)$ is the boundary stiffness matrix (also known as the Schur complement matrix), $\tilde{f}_B = \sum_i^{n_\Omega} \left(f^i_B - A^i_{BI} A^{i^{-1}}_{II} f^i_I \right)$ is the right-hand vector.

In practice, an improved algorithm of the substructuring method is used. Since the matrices corresponding to the subdomains are nonsingular and positive definite, there exists a Cholesky decomposition $A_{II} = L_{II} L^T_{II}$, where L is a lower triangular matrix with positive diagonal elements. Cholesky factorization significantly reduces the computational cost and memory requirements.

In Algorithm 1, we present a sequential algorithm of the Schur complement method (the number of processors $n_p = 1$ and $n_\Omega > n_p$) and analyze its computational cost. While counting operations at each step (given in brackets), we take into account the symmetry of matrices and consider addition and multiplication as a single operation.

Algorithm 1. *Sequential algorithm of the Schur complement method*:

1. Calculate the Cholesky decomposition of the matrix A_{II} for the subdomain i (superscript i omitted)

$$A_{II} = L_{II} L^T_{II} \qquad \left(n_I^3/6\right)$$

2. Compute

$$A'_{IB} = A^{-1}_{II} A_{IB} \qquad \left(n_B \cdot n_I^2\right)$$

3. Form the Schur complement matrix

$$S_{BB} = A_{BB} - A_{BI} A'_{IB} \qquad \left((n_I \cdot n_B^2)/2\right)$$

4. Form the right-hand vector

$$\tilde{f}_B = f_B - A_{BI} A^{-1}_{II} f_I \qquad \left(n_I \cdot n_B\right)$$

5. Form and solve the system of equations

$$S_{BB}\tilde{u}_B = \tilde{f}_B \qquad (k(M^{-1}S_{BB}) \leq C(1 + \log(H/h)))$$

6. Find the unknowns corresponding to the interior nodes

$$u_I = A^{-1}_{II} f_I - A'_{IB}\tilde{u}_B \qquad \left(n_I^2\right)$$

Here $n_I = m \cdot |\hat{V}_{I_k}|$, $n_B = m \cdot |\hat{V}_{B_k}|$ are the numbers of the interior and boundary degrees of freedom, m is the number of degrees of freedom in a node of the mesh, h is the mesh size, H is the size of the subdomain, M is the preconditioner for the Schur complement.

At step 5 of Algorithm 1, instead of the computational cost, which depends on the linear solver used, we estimate the condition number of the matrix S_{BB}. S_{BB} is considerably smaller than the initial matrix, but it is still too large for the system $S_{BB}\tilde{u}_B = \tilde{f}_B$ to be solved by direct methods. Therefore, iterative methods and efficient matrix storage schemes are required. Since the matrices S_{BB}, A_{II} are positive definite and symmetric, the preconditioned conjugate gradient (PCG) method can be used.

In this work, we present a hybrid implementation of the Schur complement method, combining different matrix storage formats, PCG solvers, and heterogeneous computing devices.

3 Analysis of Matrix Storage Formats for the Schur Complement Method

In this section, we analyze different matrix storage schemes for the Schur complement method. The matrices S_{BB}, A_{II} are symmetric, so they can be stored more compactly if only the relevant triangle is packed by columns in a one-dimensional array (upper or lower triangular format). The following analysis of the matrix storage schemes will take into account the symmetry. The full format, when all matrix elements $N(N+1)/2$, including zeros, are stored, is the most memory-consuming format. It is inappropriate for large problems.

In the matrix storage scheme DCSR implemented in the framework for finite-element analysis FEStudio [7,8], a matrix is stored in a list of packed rows. A packed row is represented by two arrays. The first array contains non-zero elements of the matrix, the second array contains their column indices in the row. The size of each array for the row i is equal to the number of non-zero elements Nnz_i and changes dynamically if necessary. The proposed matrix format DCSR is a dynamic variant of a commonly used sparse-matrix storage format CSR (Compressed Sparse Row) [9]. For a matrix A in the CSR format, three one-dimensional arrays are allocated: the first array contains non-zero values $\{a_{ij} \,|\, a_{ij} \neq 0\}$, $1 \leqslant i, j \leqslant Nnz$, the second array contains their column indices $\{j \,|\, a_{ij} \neq 0\}$, $1 \leqslant i, j \leqslant Nnz$, and the third array contains the indices of the elements a_{i1}, $1 \leqslant i \leqslant N+1$, in the first two arrays. Here N is the size of the matrix, Nnz_i is the number of non-zero elements in i-th row of the matrix, $Nnz = \sum_{i=1}^{N} Nnz_i$ is the number of non-zero elements in the matrix.

In Table 1, we compare the DCSR, CSR and the banded matrix storage formats in terms of the algorithmic complexity of search and insert, memory requirements, access rate. For the banded format, the complexity of search is $\mathcal{O}(2N_\beta + 1)$ and the complexity of insertion is $\mathcal{O}((2N_\beta + 1)N)$. For the DCSR format $\mathcal{O}(Nnz^*)$ and $\mathcal{O}(Nnz^*)$ respectively. For the CSR format $\mathcal{O}(Nnz^*)$ and $\mathcal{O}(Nnz)$ respectively. Here $N_\beta = \max_{i=1}^{N}\{\max_{j=1}^{N}\{j - i \,|\, a_{ij} \neq 0\}\}$ is a so-called half-bandwidth, which depends on the enumeration of unknowns and equations, $Nnz^* = \max_{i=1}^{N}\{Nnz_i\}$. For symmetric matrices, the values $2N_\beta + 1$ should be replaced by $N_\beta + 1$ in the complexity estimates, and Nnz and Nnz_i will correspond to the non-zero elements of the relevant triangle. The banded storage scheme requires $8(2N_\beta+1)N$ bytes of memory, the DCSR format $\sum_{i=1}^{N}(12Nnz_i+4)$ bytes, the CSR format $12Nnz+4(N+1)$ bytes. Here we assume that 8 bytes are allocated for floating-point values, and 4 bytes for integer values.

Table 1. Comparison of matrix storage formats

Format	Search	Insert	Memory	Access rate
banded	$\mathcal{O}(2N_\beta + 1)$	$\mathcal{O}((2N_\beta + 1)N)$	$8(2N_\beta + 1)N$	$\mathcal{O}((2N_\beta + 1)N)$
DCSR	$\mathcal{O}(Nnz^*)$	$\mathcal{O}(Nnz^*)$	$\sum_{i=1}^{N}(12Nnz_i + 4)$	$\mathcal{O}(Nnz)$
CSR	$\mathcal{O}(Nnz^*)$	$\mathcal{O}(Nnz)$	$12Nnz + 4(N + 1)$	$\mathcal{O}(Nnz)$

The low cost of the search and insert operations provided by the banded format is spoiled by its poor parameters of required memory, $8(2N_\beta+1)N$ bytes, and access rate, $\mathcal{O}((2N_\beta + 1)N)$, versus $\mathcal{O}(Nnz)$, the access rate of DCSR and CSR.

The DCSR format is more convenient than the CSR format for the triangular factorization of the matrices A_{II}. Indeed, when a new non-zero element is inserted in a middle of a packed DCSR row, there is no need to shift the following elements in the array, as it occurs in CSR. Therefore, insertion in DCSR has a lower algorithmic complexity, $\mathcal{O}(Nnz^*)$, than in CSR, $\mathcal{O}(Nnz)$. In addition, the memory requirement of the DCSR format, $\sum_{i=1}^{N}(12Nnz_i + 4)$ bytes, is minimal among all formats. However, direct matrix copy in the DCSR format from host to GPU memory is very expensive, because it requires a number of operations implementing the row-by-row copying. In total, N memory allocations and copying of two arrays of the size Nnz_i are required. Therefore, the CSR format is more appropriate for GPU; it requires allocation and copying of only two arrays of the size Nnz.

In [10], it was shown that the matrix storage format significantly affects the execution time. For band matrices, the execution time to construct the Schur complement is about 80% of the total time. Conversion of matrices A_{BB}, A_{IB}, A_{II} to the DCSR format reduces the execution time four times. Therefore, the serial implementation of the Schur complement method will be efficient in terms of memory requirements and algorithmic complexity when the Cholesky decomposition of A_{II} is performed and the compressed matrix storage scheme is used.

In conclusion, multiple matrix formats are required for efficient implementation of hybrid solvers. This will incur extra cost related to conversion of matrices from one format to another.

4 The Efficiency of the Parallel Schur Complement Method

The Schur complement method can be parallelized at several levels. First, computations are distributed between the subdomains [8,10], then, within individual subdomains. In addition, solving the Schur complement system (interface system) can be performed in parallel. In this paper, we mainly consider the second level of parallelization (within subdomains), and propose several algorithms to form the Schur complement matrices and to solve the interface system of equations with multiple graphical accelerators.

Let us consider Algorithm 1 in terms of parallelization. Steps 1–4 and 6 are performed for each subdomain independently, therefore we can say that there is a natural parallelism at the first level. The most computationally intensive steps of the algorithm are the following:

– steps 1–3, where the Schur complement matrix is formed,
– step 4, where the right-hand vector is formed, and
– step 5, where the system of equations is solved.

Step 3, where the local matrices for subdomains S^i_{BB} are formed, can be performed in parallel. However, step 5, where the system of equations with the global Schur complement matrix S_{BB} is solved, cannot be carried out until all parallel processes on subdomains complete construction of their local stiffness matrices.

There are two main sources of unbalance. One is *nonuniform distribution of subdomains between computational nodes*. This problem can be resolved easily by partitioning the domain into the number of subdomains that is a multiple of the number of computational nodes. Another source of unbalance is *nonuniform distribution of the mesh nodes and finite elements between subdomains*, which occurs, for example, due to mesh refinement [11]. To balance the load in this case, additional conditions are imposed on mesh partitioning. Load balancing of steps 1–4 depends on the number of interior $|\hat{V}_{I_i}|$ and boundary $|\hat{V}_{B_i}|$ nodes in the subdomains Ω_i, rather than on the total number of nodes $|\hat{V}_i|$ in those subdomains.

Meshes for 3D domains with a complex surface (springs, thin plates and shells) contain a lot of elements with all nodes belonging to the boundary of the mesh. Partitioning the dual graphs of these meshes and enforcing the condition $|V_i| \approx |V_j|$, $\forall i \neq j$ result in the subdomains Ω_i: $|\hat{V}_{I_i}| = 0$. For illustration, we consider the problem of stress-strain analysis of a wrap spring. In a wrap spring, the number of boundary nodes is greater than the number of interior nodes. This case is extreme for computing the Schur complement. Since the size of the matrix S_{BB} is larger and its sparsity is lower, solving the system of equations (3) takes longer than forming this system.

Let us consider a spring approximated by an unstructured tetrahedron mesh (see Fig. 1a), where the number of tetrahedrons is $|V| = 174264$, the number of nodes $|\hat{V}| = 40743$. To partition this mesh into subdomains Ω_i : $|\hat{V}_{I_i}| > 0$, the weighted dual graph $G(V, E, W)$ is built, with the weights $W = \{w_k\}$. If at least one node of a finite element does not belong to $\partial\Omega$, then the weight of the corresponding graph node will be $w_k = 3$, otherwise we assign $w_k = 1$.

The experiments with the number of the subdomains ($n_\Omega = 16, 32, \ldots, 1024$) show that the larger the number of the substructures, the larger the size of the Schur complement matrix. If the number of subdomains is increased in 64 times, then the size of the matrix S_{BB} will be increased in 1.44 times ($N = 66030$ if $n_\Omega = 16$, and $N = 95523$ if $n_\Omega = 1024$). At the same time, the number of nonzero elements of the Schur complement matrix decreases in 18 times (from $Nnz \approx 2.7 \cdot 10^8$ in the case of 16 subdomains to $Nnz \approx 1.5 \cdot 10^7$ for 1024 subdomains). As a result the density of the matrix decreases from 6.11% to 0.16%. In average, every sixteenth element in a row of S_{BB} is nonzero (4041 out of 66030) for $n_\Omega = 16$ subdomains (see Fig. 1b), and approximately every six hundredth (154 out of 95523) for 1024 subdomains (see Fig. 1c).

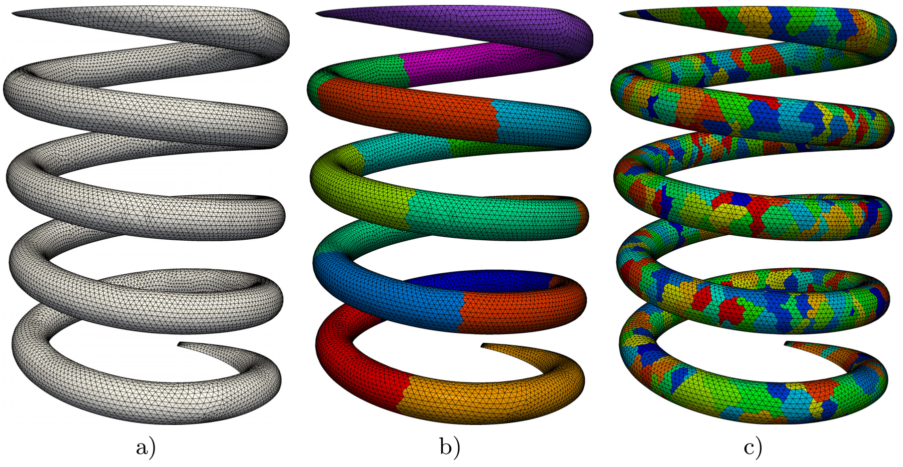


Fig. 1. Mesh: a) initial; b) divided into 16 subdomains; c) divided into 1024 subdomains by the two-level partition.

The size of the Schur complement matrix is less than the size of initial finite element system (in our examples in 1.4–2.0 times). The density of the matrix S_{BB} is of one or two orders of magnitude higher than the density of the global stiffness matrix (0.03%). For S_{BB} of such density, graphical accelerators can be efficiently used to solve the interface system. For massive bodies, solution of the interface system is less expensive in comparison to construction of the system (3).

4.1 Construction of the Schur Complement Matrix

The inversion of the matrix A_{II} is one of the most expensive operations in the construction of the Schur complement matrix. Usually, to inverse the matrix, inefficient direct methods are used, for example, LL^T-factorization. In this work, we propose an inversion algorithm that solves the system $A_{II}X = E$, where E is the identity matrix $n_I \times n_I$. This system can be solved efficiently on GPU, using the preconditioned conjugate gradient method.

If we substitute A_{IB} for E into the right-hand part, then the solution of the system will be $A^{'}_{IB} = A^{-1}_{II} A_{IB}$. Matrix inverse and multiplication can be replaced by solving n_B systems independently, and hence, several GPU can be used in parallel. To compute the Schur complement matrix, we use (3), where $A_{BB} \in \mathbb{R}^{n_B \times n_B}$, $A_{II} \in \mathbb{R}^{n_I \times n_I}$ $A_{BI} \in \mathbb{R}^{n_B \times n_I}$, $A_{IB} \in \mathbb{R}^{n_I \times n_B}$. Let $\tilde{n}^i_B$ be the number of columns for the matrix A_{BB} being sent to i-th GPU, $\tilde{A}^i_{BB} \in \mathbb{R}^{n_B \times \tilde{n}^i_B}$ be the matrix made by the columns of the matrix A_{BB} with the numbers from $\sum_{j=0}^{i} \tilde{n}^j_B$ to $\sum_{j=0}^{i+1} \tilde{n}^j_B$, where i is the GPU identifier. Each GPU solves $\tilde{n}^i_B$ systems $A_{II} a^k = a^k_{IB}$, where a^k_{IB} is the k-th column of the matrix A_{IB}, $k \in \left[\sum_{j=0}^{i} \tilde{n}^j_B, \sum_{j=0}^{i+1} \tilde{n}^j_B\right]$ and $A^{'}_{IB} = \{a^1, a^2 \ldots a^{n_B}\}$.

To solve the systems of equations independently on several GPU on a single computing module, we use OpenMP. Several threads are spawned and assigned to GPUs. The number of threads is equal to the number of available GPUs. Multi-GPU construction of the Schur complement is performed as follows.

Algorithm 2. *A parallel algorithm for the forming of Schur complement for each subdomain Ω_i* (superscript i is omitted)

```
1. Solve the system $A_{II} A^{'}_{IB} = A_{IB}$ on GPU;
   {matrices are stored in the CSR format, the solution is stored in columns}
2. $S_{BB} = A_{BB} - A_{BI} A^{'}_{IB}$;
   {$S_{BB}$ and the result of $A_{BI} A^{'}_{IB}$ are stored as a set of rows, $A_{BB}$ is stored
   in the CSR format}
3. Form $\tilde{f}_B = f_B - A_{BI} x$; {$x$ is the solution of $A_{II} x = F_I$}
4. Form and solve: $S_{BB} \tilde{u}_B = \tilde{f}_B$;
   {$S_{BB}$ is formed in the DCSR format on the CPU, and then converted into
   the CSR format on GPU}
5. Define $u_I = x - A^{'}_{IB} \tilde{u}_B$;
   {here $x$ and $A^{'}_{IB}$ have been computed earlier and stored on the CPU}
```

After the system has been solved, the matrix $(A^{'}_{IB})^i \in \mathbb{R}^{n_I \times n^i_B}$ is stored on each GPU. It consists of the columns of the matrix $A^{'}_{IB}$ with numbers from $\sum_{j=0}^{i} \tilde{n}^j_B$ to $\sum_{j=0}^{i+1} \tilde{n}^j_B$. The rest of computations (product and subtraction of the matrices, see (3)) can be performed without additional communications for every matrix $(A^{'}_{IB})^i$, independently on each GPU. Finally, the global Schur complement matrix S_{BB} is formed (step 4 in Algorithm 2) as a sum of local matrices S^i_{BB} for the i-th subdomain. The local Schur complement matrices S^i_{BB} are stored in the general uncompressed format. The global matrix S_{BB} is converted from the uncompressed format to the format appropriate for GPU, such as CSR.

In Table 2, the results of experiments to form the Schur complement are presented for different numbers of subdomains and GPUs (Algorithm 2).

In the second column, the cost for a sequential algorithm on CPU is shown (Algorithm 1). The speedup of parallel Algorithm 2 on mulitple GPUs relatively to a single CPU can be defined as $s(n_p)_{CPU} = t_{CPU}/t(n_p)_{GPU}$, where t_{CPU} is the time of the corresponding sequential algorithm on CPU, and $t(n_p)_{GPU}$ is the time of the parallel algorithm on n_p GPUs. Similarly, we define the speedup relatively to a single GPU: $s(n_p)_{GPU} = t(1)_{GPU}/t(n_p)_{GPU}$.

Table 2. Time to form the Schur complement, sec.

n_Ω	CPU	1 GPU	2 GPU	4 GPU	6 GPU	8 GPU
16	1583.6	1912.6	1165.5	789.9	657.6	589.4
32	480.6	1173.9	661.8	401.7	314.0	266.5
64	145.1	678.8	378.2	224.8	174.7	149.0
128	—	—	237.2	145.8	118.5	102.5
256	—	—	194.0	121.0	97.7	86.0
512	—	—	168.3	105.7	86.0	75.4
1024	18.7	233.4	144.0	92.2	77.5	68.0

The maximum speedup relatively to GPU $s(8)_{GPU} = 2.7$ was observed for the number of subdomains $n_\Omega = 16$, with the average number of elements 11000. When the Schur complement matrix was formed on two GPU, the speedup was $s(2)_{GPU} > 1.5$, depending on the number of subdomains. Eight GPU gave the least execution time, but for the problems with the small numbers of elements in a subdomain (< 2500), CPU was faster. In this case, due to solving a large number of small systems of equations for each subdomain, communications between GPU and CPU take longer than solution of the system. When multiple GPUs are used, the cost per one GPU is reduced, but it does not cover the cost of initialization and communications.

Subdomain computations are distributed between computing modules accordingly to the partitioned mesh distributed between MPI processes [12]. Introduction of this parallelization level speeds up construction of the Schur complement proportionally to the number of computing modules.

4.2 Solution of the Schur Complement System on GPU

The Schur complement matrix $S_{BB} \in \mathbb{R}^{N \times N}$ (below, it will be denoted as S) is symmetric, positive-definite, sparse, and has the size and the condition number smaller than those of the original matrix. Solution of the linear algebraic equations (3) and the systems of equations from the previous section can be found by the preconditioned conjugate gradient method (PCG). The optimal choice for GPU computing is the preconditioners $\bar{M}$ based on the approximation of the inverse matrix of the system [13]. In this case, the additional operations to get the preconditioned system can be reduced to the matrix-vector product $z_{k+1} = \bar{M} r_{k+1}$.

We solve (3) by using the conjugate gradient method implemented with help of CUDA as presented in

Algorithm 3. *Preconditioned conjugate gradient method on GPU* :

```
1. Initialization (1a-1g):
   (a) $S, \bar{M} \in \mathbb{R}^{N\times N}$ {$\bar{M}$ is formed on GPU and is stored in the CSR format}
   (b) $u, r, p, q, z \in \mathbb{R}^N$ {vectors are stored only in GPU memory}
   (c)   $r_0 \leftarrow f$ {cublasDcopy}
   (d) $u_0 \leftarrow 0$ {initialization on GPU}
   (e) $z_0 \leftarrow \bar{M} r_0$ {performed on GPU}
   (f) $p_0 \leftarrow z_0$ {cublasDcopy}
   (g) $\rho_0 \leftarrow (r_0, z_0)$ {here and below $(\cdot,\cdot) = \sum_P (\cdot,\cdot)^{(P)}$}
2. Iterative process.
   While $||r_i||_2/||b||_2 > \varepsilon$ do (2a-2h):
   (a) $q_i \leftarrow S p_i$ {performed on GPU}
   (b) $\alpha_i \leftarrow (r_i, z_i)/(q_i, p_i)$ {cublasDdot}
   (c) $u_{i+1} \leftarrow u_i + \alpha_i p_i$ {cublasDasxpy}
   (d) $r_{i+1} \leftarrow r_i - \alpha_i q_i$ {cublasDasxpy}
   (e) $z_{i+1} \leftarrow \bar{M} r_{i+1}$ {performed on GPU}
   (f) $\rho_{i+1} \leftarrow (r_{i+1}, z_{i+1})$ {cublasDdot}
   (g) $\beta_{i+1} \leftarrow \rho_{i+1}/\rho_i$
   (h) $p_{i+1} \leftarrow z_{i+1} + \beta_{i+1} p_i$
       {consecutive invocations of cublasDscal and cublasDasxpy}
```

All auxiliary arrays, including r, p, q, z, and the matrix of the system, preconditioner, the right-hand vector, and the solution vector are stored in the memory of GPU. After the conjugate gradient method is finished, array u, which stores the approximation of the solution vector, is copied to the memory of CPU. We use the CUBLAS library to implement the following operations: sum, inner product, vectors copy, and scalar vector multiply. When performing the matrix-vector product, the vector is stored in the texture memory, which is cached, providing a faster access time. To calculate vector coordinates, from 2 to 32 threads are used, depending on the sparsity of the matrix. Preconditioner $\bar{M}$ is computed on GPU or CPU in accordance with its type.

We compared our PCG solver with its counterpart implemented the CUSP library (`http://cusplibrary.github.io/`), using a single GPU. Matrices and right-hand-side vectors were obtained from the test problem for finite-element elasticity simulation with basis functions of different types. Figure 2 shows the patterns of the matrices. Performance comparison of PCG solvers is presented in Table 3. Our solver outperforms CUSP for small problem sizes due to the use of CUBLAS instead of Thrust. For large problem sizes, performance of both solvers is similar.

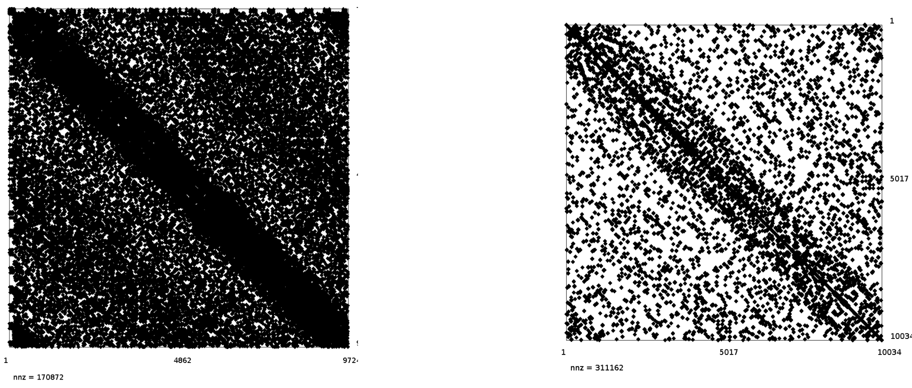


Fig. 2. Sparse matrices patterns

Table 3. Performance comparison of PCG (our implementation vs CUSP), sec.

N	Nnz	Our	CUSP	N	Nnz	Our	CUSP
952	15856	0.009	0.043	1042	30170	0.353	1.742
2426	72770	0.484	2.156	2568	44164	0.017	0.075
4960	86344	0.027	0.106	4758	145286	0.761	3.115
9724	170872	0.048	0.160	10034	311162	1.402	4.763
100660	1798378	0.817	1.082	100886	3196070	31.57	40.20
496842	15827426	82.93	85.96				

4.3 Solving the Interface System on Multi-GPU and Cluster-GPU

To solve the interface system (3) on Multi-GPU, a block algorithm of the conjugate gradient method was implemented, with computations distributed between n_p GPUs with help of OpenMP. In this case, Algorithm 3 is implemented in a parallel region, created by the directive `omp parallel`. The matrix $S = \{s_{ij}\}$ of (3) is partitioned into blocks. The results of the our PCG inner products in separate threads of OpenMP are stored in shared variables and summarized by the directive `atomic`, followed by a barrier synchronization.

To divide the matrix S, we represent it as a graph $G_S(V, E)$, where $V = \{i\}$ is a set of vertices, which correspond to the row indices (the number of vertices is equal to the dimension of S); $E = \{(i, j)\}$ is a set of edges, whose ends correspond to the row and column indices of nonzero elements of S. A graph G_S is divided into n_p parts by the multilevel algorithm [14]. After that, every vertex of the graph is assigned to the GPU identifier $k \in [1, n_p]$. According to their GPU identifiers, the vertices are divided into the internal and boundary vertices. The latter are associated with at least one vertex that has a different GPU identifier.

After partitioning, each block S_k contains several matrices:

– $S_k^{[i_k,i_k]}$ is the matrix associated with the internal vertices;
– $S_k^{[i_k,b_k]}$, $S_k^{[b_k,i_k]}$ are the matrices associated with the internal and boundary vertices;
– $S_k^{[b_k,b_m]}$ is the matrix associated with the boundary vertices of the k-th and m-th blocks.

Here $k \neq m$ and $k, m \in [1, n_p]$, and n_p is the number of blocks.

Matrix S can be rewritten as follows

$$S = \begin{pmatrix} S_1^{[i_1,i_1]} & S_1^{[i_1,b_1]} & \cdots & 0 & 0 & \cdots & 0 & 0 \\ S_1^{[b_1,i_1]} & S_1^{[b_1,b_1]} & \cdots & 0 & S_1^{[b_1,b_k]} & \cdots & 0 & S_1^{[b_1,b_{n_p}]} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & S_k^{[i_k,i_k]} & S_k^{[i_k,b_k]} & \cdots & 0 & 0 \\ 0 & S_k^{[b_k,b_1]} & \cdots & S_k^{[b_k,i_k]} & S_k^{[b_k,b_k]} & \cdots & 0 & S_k^{[b_k,b_{n_p}]} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & S_{n_p}^{[i_{n_p},i_{n_p}]} & S_{n_p}^{[i_{n_p},b_{n_p}]} \\ 0 & S_{n_p}^{[b_{n_p},b_1]} & \cdots & 0 & S_{n_p}^{[b_{n_p},b_k]} & \cdots & S_{n_p}^{[b_{n_p},i_{n_p}]} & S_{n_p}^{[b_{n_p},b_{n_p}]} \end{pmatrix}.$$

When the matrix S is multiplied by the vector p, two vectors are computed on each GPU:

$$q_k^b = S_k^{[b_k,i_k]} p_k^i + \sum_{m=1}^{m \leqslant n_p} S_k^{[b_k,b_m]} p_m^b \qquad q_k^i = S_k^{[i_k,i_k]} p_k^i + S_k^{[i_k,b_k]} p_k^b, \tag{4}$$

where k is a GPU identificator, and $p^T = \left(p_1^i, p_1^b, \ldots, p_k^i, p_k^b, \ldots, p_{n_p}^i, p_{n_p}^b\right)$. This implementation of the matrix-vector product reduces communications between the blocks at each iteration of the conjugate gradient method. Indeed, to perform the following steps, the exchange of vectors q_k^b is required, whose size is much smaller than that of the vector q.

The results in Table 4 show that the GPU algorithm of the conjugate gradient method significantly speeds up solution of the interface system. For 16 subdomains, the speedup was $s(1)_{CPU} = 72$. For 1024 subdomains, $s(1)_{CPU} = 94$. Multiple GPUs provided the speedup $s(8)_{CPU} = 251$ for $n_\Omega = 64$ and $s(8)_{GPU} = 3.5$ for $n_\Omega = 16$. The increase in the number of subdomains results in reducing the number of nonzero elements in the Schur complement matrix. Thus, the efficiency of using multiple GPUs to solve the system (3) is reduced. For example, the speedup was $s(8)_{GPU} = 1.3$ for 1024 subdomains.

In our experiments, the minimal total time to form and solve (3) was obtained in the case $n_\Omega = 1024$ (see Tables 2 and 4). It took 1 hour and 48 minutes to perform these steps on a CPU only. In the case of a single GPU, the cost of construction and solution of this system was reduced in 22 times. The minimal time computing $t(8)_{GPU} = 2$ min was reached, when eight GPU were involved. It took one and a half minute to form the system (3) on CPU and to solve it on

Table 4. The time to solve (3) in a one computational module, sec.

n_Ω	CPU	1 GPU	2 GPU	4 GPU	6 GPU	8 GPU
16	> 28800	912	421	205	281	264
32	> 28800	983	630	430	317	274
64	18067	287	174	98	82	72
128	—	—	127	78	66	60
256	—	—	106	70	61	56
512	—	—	85	63	56	53
1024	6502	69	76	59	54	52

GPU. We obtained the lowest total cost, when the system was formed on CPU and then was solved on eight GPUs.

The system of equations (3) can also be solved on several computational modules with GPU (Cluster-GPU). We made experiments with the following configurations:

- eight MPI processes, performed in a single module (1×8);
- four processes in two modules (2×4);
- two processes in four modules (4×2), and
- one process in eight modules (8×1).

For communications, we used MPI functions: Allgatherv to assemble the vector q and Allreduce to sum scalars α, β, ρ. These functions were called by one of the OpenMP threads, running inside each MPI process.

Experiments show the execution time of (3) significantly depends on distribution of computations of matrix vector product between CPU and GPU. In case of $n_\Omega = 16$ (see Table 5), the most computationally expensive part is $\hat{q}_k^b = \sum_{m=1,m\neq k}^{m<n_p} S_k^{[b_k,b_m]} p_m^b$ of (4), which is computed on CPU. This is caused by the large sizes of blocks of $S_k^{[b_k,b_m]}$ and by their uneven distribution between parallel processes/threads. In such cases, it is possible to transfer this operation to GPU. The results given in Table 5 show that distribution of computations between different computing modules results in acceleration of 1.2–1.5 times, most likely due to the competition for CPU cache memory.

More uniform partition of G_S subgraphs boundary vertices and smaller sizes of $S_k^{[b_k,b_m]}, m \neq k$ lead to more efficient computation of $\hat{q}_k^b$ for CPU for $n_\Omega = 1024$, and therefore, to domination of vector operations in the execution time (see Algorithm 3, 2b, 2f). In this case, invocations of `cublasDdot` at scalar values take about 90% of execution time (variants 1×8, 2×4, 4×2).

Acceleration of calculations with a larger number of subdomains results from a more sparse systems of equations (Nnz reduced, N increased). Therefore, the total number of arithmetic operations in the multiplication of matrix S_{BB} and vector p (3) is decreased. Distribution of matrix blocks between computing

Table 5. The time to solve (3) in 1 to 8 computational modules, sec.

n_Ω	N	NNz	1×8	2×4	4×2	8×1
16	66030	266826696	321	268	208	233
64	70620	90931750	78	72	71	95
128	75732	58592886	68	60	59	86
256	81753	38066007	66	57	66	87
512	88248	23955996	63	55	68	102
1024	95523	14749329	65	54	54	110

modules allows us to remove the restriction on the size of the interface system (3), but requires further improvement of communication algorithms.

We do not provide comparison of this solver with CUSP, because CUSP does not support multi-GPU computations. The presented results were obtained from the hybrid cluster Uranus in the IMM UB RAS, which consists of dual processor nodes Intel Xeon E5675 with eight GPUs NVIDIA Tesla M2090.

5 Conclusion

The hybrid implementation of the Schur complement method presented in this paper allowed us to distribute computation between CPU and GPU in a balanced way. The optimal choice of the algorithm to form the Schur complement depends on the number and size of subdomains into which the mesh is divided. If one subdomain contains relatively small number of mesh elements (< 5000) or unknowns (< 1500 for internal and < 2500 for the boundary nodes), then it is more efficient to use direct methods to find the inverse matrix, and therefore, to use only CPU. For large problems, iterative algorithms executed on several GPUs are the most efficient. The interface system of equations should be solved on GPUs, which can accelerate this step in tens and hundreds of times.

Acknowledgments. This research is supported by RFBR (projects: 11-01-00275-a, 12-07-31114-mol_a) and the joint program N18 of the Presidium of RAS and the Ural Branch of RAS (project 12-P-1-1005).

References

1. Giraud, L., Haidar, A., Saad, Y.: Sparse approximations of the Schur complement for parallel algebraic hybrid solvers in 3D. Numerical Mathematics 3, 276–294 (2010)
2. Gaidamour, J., Henon, P.: A parallel direct/iterative solver based on a Schur complement approach. In: IEEE 11th International Conference on Computational Science and Engineering, Sao Paulo, Brazil, pp. 98–105 (2008)

3. Agullo, E., Giraud, L., Guermouche, A., Roman, J.: Parallel hierarchical hybrid linear solvers for emerging computing platforms. Comptes Rendus Mecanique 339(2-3), 96–103 (2011)
4. Yamazaki, I., Li, X.S.: On techniques to improve robustness and scalability of a parallel hybrid linear solver. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 421–434. Springer, Heidelberg (2011)
5. Rajamanickam, S., Boman, E.G., Heroux, M.A.: Shylu: A hybrid-hybrid solver for multicore platforms. In: IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS, pp. 631–643 (2012)
6. Przemieniecki, J.: Theory of Matrix Structural Analysis. McGaw-Hill, New York (1968)
7. Kopysov, S.P., Krasnoperov, I.V., Rychkov, V.N.: An object-oriented method for domain decomposition. Numerical Methods and Programming 4, 176–193 (2003)
8. Kopysov, S.P., Krasnopyorov, I.V., Novikov, A.K., Rychkov, V.N.: Parallel Distributed Object-Oriented Framework for Domain Decomposition, pp. 605–614. Springer (2006)
9. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM (2003)
10. Kopysov, S.: Optimal domain decomposition for parallel sustructuring method. In: Mesh Methods for Boundary-Value Problems and Applications. Proceedings of 5th Russian Seminar, pp. 121–124. Kazan University, Kazan (2004)
11. Kopysov, S.P., Novikov, A.K.: Parallel adaptive mesh refinement with load balancing on heterogeneous cluster, pp. 425–432. Nova Science Publishers (2006)
12. Kopysov, S.P., Krasnoperov, I.V., Rychkov, V.N.: Implementation of an object-oriented model of domain decomposition on the basis of parallel distributed corba-components. Numerical Methods and Programming 4(1), 19–36 (2003)
13. Kopysov, S.P., Novikov, A.K., Sagdeeva, Y.A.: Solving of discontinuous galerkin method systems on gpu. Bulletin of Udmurt University. Mathematics. Mechanics. Computer Science (4), 121–131 (2011)
14. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. SIAM Review 41(2), 278–300 (1999)

Formal Verification of Programs in the Pifagor Language

Mariya Kropacheva and Alexander Legalov[⋆]

Siberian Federal University, Institute of Space and Information Technology,
26 Kirenskogo Street, Krasnoyarsk, 660074, Russia
ksv@akadem.ru, legalov@mail.ru

Abstract. The article is devoted to the methods of proving parallel programs correctness, that are based on the Hoare axiomatic system. In this article such system is being developed for proving the correctness of the programs in the functional data-flow parallel programming language Pifagor. Recursion correctness is proved by induction. This method could be used as a base of a toolkit to support program correctness proving, since it could be made automate at many stages.

Keywords: Functional data-flow parallel programming, Pifagor programming language, programs formal verification.

1 Introduction

Nowadays computers are widely used in safety critical systems so it is extremely important to develop reliable software. Formal verification could be used to increase software reliability. Formal verification is a proof of programs correctness by finding the correspondence between the program and its specification, which describes the aim of the development [1]. The main advantage of formal verification is the capability to prove the absence of errors in the program, while testing only allows to detect errors.

One method of formal verification was introduced by Hoare [2]. It utilises an axiomatic approach based on Hoare logic. Hoare logic is an extension of a formal system $\mathfrak{I}$ with certain formulas called Hoare triples. A Hoare triple is an annotated program, namely the source code and two formulas of the theory $\mathfrak{I}$, which describe restrictions on input variables and correctness conditions of the result of the program execution. These formulas are called precondition and postcondition, respectively. The extended formal system is distinguished from $\mathfrak{I}$ by additional axioms and inference rules, which allow to deduce certain program properties, particularly the program correctness. The program is correct if its Hoare triple is identically true. So the main idea of this approach is to derive a formula of formal system $\mathfrak{I}$ from the Hoare triple and then prove the truth of this formula within the formal system $\mathfrak{I}$.

[⋆] This work was supported by the Federal Targeted Program "Scientific and Scientific-Pedagogical Personnel of Innovative Russia" under the grant No.14.A18.21.0396.

There are certain achievements in practical application of such an approach for imperative programming languages [1], [3]. However formal verification complexity for parallel imperative programs increases rapidly for systems with both shared and distributed memory. In general, the main problem is the system resource conflicts. An alternative to imperative programming is functional data-flow paradigm for parallel programming in which a program is represented as a directed data-flow graph.

One implementation of functional data-flow paradigm is the Pifagor (Parallel Informational and Functional AlGORthmic) language [5]. In Pifagor there are neither variables nor looping constructs, and function execution starts only on data readiness. Hence, it excludes resource conflicts. Another distinguishing feature of this language is the ability to achieve the maximum parallelism of the program, as parallelism is implemented at the level of operations. So after formal verification the correct program could be transferred to the system with specific architecture and limited resources, with the reduction of parallelism if needed. It makes the complexity of formal verification of functional data-flow parallel programs equal to the complexity for sequential programs. Currently Pifagor is used in research of development principles of architecture-independent parallel programs.

Nowadays there exist some works dedicated to functional data-flow programs debugging [4] but formal verification problem is not developed. So the development of the formal verification methods for functional data-flow programs is topical.

2 The Description of the Formal System

The formal system for the functional data-flow parallel language Pifagor is needed to perform formal verification of program correctness based on the axiomatic approach. It is necessary to define language (an alphabet, formation rules), axioms and inference rules.

The Hoare triples are the main objects of the formal system. Preconditions and postconditions are formulas in first-order logic, which is expressive enough for most assertions of the program.

The alphabet of first order logic, functional and predicate symbols, corresponding to functions of the programming language, are used for constructing expressions. Domain variables (input and output program variables) could be of different types corresponding to the types of the programming language. Functions, which together with variables form terms, and predicates, which form formulas, are distinguished in first-order logic. It is not necessary to separate predicates from formulas in this case. Predicates could be considered as a subset of functions with the range of values from the set *bool*. Let us introduce certain functional and predicate symbols: arithmetic operations $(+, -, *, /)$, relational symbols $(=, \neq, >, <, \geqslant, \leqslant)$, logical operators and quantifiers $(\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow, \forall, \exists)$, "length of the list" function (*len*), "select list element" function (*select*), "is of type" function $(\in)$.

The functions *len*, *select*, $\in$ are equivalent to the corresponding built-in functions of the Pifagor language. The type signatures of the functions mentioned above correspond to those of the functions and predicates of first-order logic and arithmetic. The set of elementary terms, elementary formulas and formulas are defined inductively as in first-order logic.

Hence, having an alphabet and formation rules one can form preconditions and postconditions for the Hoare triples. The following notation is used for the Hoare triple:

$$\boxed{\varphi(x)}\,\texttt{Prog(x)}\!\rightarrow r\,\boxed{\psi(r)}\,,$$

`Prog(x)` being a program with input argument x, φ and ψ being the precondition and the postcondition for `Prog`, respectively, r denoting the result of the program execution. There are no variables in the Pifagor language, therefore introducing r is necessary for stating the postcondition.

For example, consider a function with the following code:

```
Fun << funcdef arg {
     arg:F >>  return
}
```

Let P and Q be the precondition and postcondition for this program. Then the Hoare triple has the following form:

$$\boxed{P(arg)}\,\texttt{arg:F}\!\rightarrow r\,\boxed{Q(r)}\,.$$

Since a program written in Pifagor is more demonstrable when represented as a data-flow graph, it is useful to attach a precondition and a postcondition to the edges of this graph. The example of the above triple in the form of data-flow graph is shown in the figure 1.

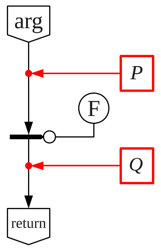


Fig. 1. A Hoare triple for program `arg:F`

Axioms of the considered formal system are Hoare triples for build-in functions. These triples are true by definition and are formed on the basis of operational semantic rules of the Pifagor language [5], [6]. It should be pointed out that the number of different paths of the function execution equals the number of axioms for this function so there may exist more that one axiom for one build-in function.

The inference rules allow to bind axioms, based on the built-in functions, to an arbitrary program. There are two alternative ways to prevent the ambiguity of the inference:

1. Using the rules of *forward tracing*, when inference rules are applied form top to bottom of the data-flow graph (from input values towards the result).
2. Using the rules of *backward tracing*, when inference rules are applied from bottom to top of the data-flow graph (from the result towards input values).

Consider the first alternative. By applying the rules of forward tracing, it is possible to transform any Hoare triple, when we start with a function applied directly to input value, namely the function that is executed on the first step. There could be more than one function in case of the Pifagor language. Then any of such functions could be selected. In the general case the rule of forward tracing has the following form:

$$\left\{ \boxed{P_1(x_1)}\, \mathtt{x_1{:}F} \to r \,\boxed{Q(r)}\,,\, A_1,\, A_2 \right\} \quad \vdash \quad \boxed{P(x)}\, \mathtt{x{:}F_1{:}F} \to r\, \boxed{Q(r)}\,, \tag{1}$$

$$A_1 := \boxed{\varphi(x)}\, \mathtt{x{:}F_1} \to x_1\, \boxed{\psi(x_1)}\,,\ A_2 := P(x) \Rightarrow \varphi(x),$$
$$P_1(x_1) := P(x) \wedge \varphi(x) \wedge \psi(x_1).$$

The turnstile symbol $\vdash$ indicates that if the Hoare triple on the left is true, then one on the right is also true. This means that by applying the axiom $\boxed{\varphi(x)}\ \mathtt{x{:}F_1} \to r_1\ \boxed{\psi(r_1)}$ to the function $\mathtt{F_1}$, the Haore triple (on the right) is transformed into a new triple (on the left) with a "shorter" program, the precondition $P(x)$ is replaced by $P_1(x)$, and input argument x is replaced by x_1, that is by the result of applying of the function F_1 to x.

So, sequential application of inference rules leads to the "shortening" of the program, which results in the Hoare triple with an "empty" program: $\boxed{\mathrm{P}}\ \boxed{\mathrm{Q}}$. Introduction of the following inference rule enables us to transform this Hoare triple into the first-order logic formula:

$$P \Rightarrow Q \vdash \boxed{\mathrm{P}}\ \boxed{\mathrm{Q}}\,. \tag{2}$$

Thus, using inference rules for any Hoare triple one can transform it into the first-order logic formula, the truth of which could be proved within the first-order logic. If the formula is true then the Hoare triple is also true, and therefore the program is correct.

Let us make sure that Hoare formal system with the rule of forward tracing is consistent, namely true triples allow to infer only true triples. Consider the forward tracing rule. Let A_1, A_2 and $A = \boxed{P_1(x_1)}\, \mathtt{x_1{:}F} \to r\, \boxed{Q(r)}$ be identically true. If $P(x)$ is true then by A_2, $\varphi(x)$ is true. Further, by A_1, $\psi(F_1(x))$ is true, and hence, $P_1(F_1(x))$ is true, and finally, by A, $Q(F(F_1(x))) = Q(r)$ is true. The proof for the rule (2) is tautology.

3 The Analysis of Recursion Correctness

The main problem of the program verification concerns repeatedly executed code, which in case of incorrect program could lead to infinite looping. In this case data-flow graph is infinite. The Pifagor language has no looping constructs and relies solely on recursion. The program is correct if, on the one hand, the program terminates in a finite number of steps, and on the other hand, its output result is correct.

If a program contains recursion, then the same code is called several times, and different are only its arguments. Then necessary condition of a recursion termination is the sequence of the arguments passed to the recursive function being unduplicated.

It is obligatory for a correct recursive function to have a "branch point", where the further execution path is selected. In the base cases, the result of the function is produced trivially (without recurring), and in the other (or, recursive) cases, the program recurs (calls itself). The choice of the case on each iteration is defined by a certain function on the input arguments. This function can be thought of as a counterpart of an if-else construct in an imperative programming language in the sense that the conditions of recurring or leaving the recurrent function are mutually exclusive, i.e. $\neg(\textit{recursive case} = \textit{true}) \Leftrightarrow (\textit{base case} = \textit{true})$.

Proving of the correctness of recursion may be done by induction (the proof of program termination is similar). Define the notion of the *bound function.* A bound function is a function, bounded above, that maps recursive function arguments to the set of natural numbers $\mathbb{N}$, and all arguments, for which the base case is true, are mapped to 1.

Let us consider the proof scheme of a recursive function *Rec* correctness. It is done inductively by the values of the bound function f. The basis: check whether the program is correct for argument $x = p_0$, so that $f(p_0) = 1$. The inductive step: assume that the program is correct for all arguments for which the bound function value are less than N. A parameter p_N corresponds to the number N such that $f(p_N) = N$. Then it is sufficient to show that during the $Rec(p_N)$ function execution, the function *Rec* is called recursively only with arguments p_i, $i = \overline{1,n}$, such that $f(p_i) < N$ ($p_i, i = \overline{1,n}$ must be permitted inputs of the function Rec). If $Rec(p_i), i = \overline{1,n}$ return correct results, then $Rec(p_N)$ terminates and it remains to show that $Rec(p_N)$ returns the correct result. The algorithm described above is a sufficient condition of the recursive function correctness. If we can not prove the correctness of the function, then either the program is incorrect or the bound function was not properly selected. In the latter case a new bound function is needed.

4 An Example of a Recursive Function Correctness Proof

Every program written in Pifagor is considered parallel due to its correspondence to a data-flow graph, which is invariant to parallel or sequential program execution. We demonstrate the capabilities of the data-flow graphs for program verification on a typical example, often shown for sequential programs. Let us prove the correctness of a recursive function `fact`, which calculates the factorial of x.

Source code of the function ***`fact`***

```
fact << funcdef x {
    fl << ((x,1):[<=,>]):?;
    act << (1, { (x, (x,1):-:fact ):*  } );
    return << act:fl:.;
}
```

We define the following triple for the program (by setting the precondition and the postcondition):

$$\boxed{\begin{array}{l}(x \in int) \wedge (x \geqslant 0) \wedge \\ \prod_{i=1}^{x} i \leqslant \texttt{INT\_MAX}\end{array}} \; \texttt{x:fact} \rightarrow r \; \boxed{\begin{array}{l}((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee \\ ((r = 1) \wedge (x = 0))\end{array}}, \tag{3}$$

where `INT_MAX` is the maximal integer of the *int* type. This Hoare triple provides the following specification: if the argument x is of the integer type and greater than zero, and the product of integers from 1 to x is not greater than the maximum integer of the *int* type, then after execution the function `fact` returns the product of integers from 1 to x in case $x > 0$, or 1 if $x = 0$.

The data-flow graph of the triple (3) is given in the figure 2. Edge identifiers are in gray circles, all of them should be unique.

Let us analyse the function `fact` execution. If $x = 0$ or 1 then the function returns 1, if $x > 1$ then the function `fact` is called recursively with the argument $x - 1$. The conditions $x \leqslant 1$ and $x > 1$ are checked in the right branch of the data-flow graph which has the node with function "?". Simultaneously in the left branch the list p of two elements is formed. The first element is a constant "1" and the second is a "delaylist" which is marked with a dashed line in the figure. This delay list has a subgraph with the recursive call of the function `fact`. Execution of operators in the delay list starts only after the delay release even if all the arguments are ready. The truth of $x \leqslant 1$ or $x > 1$ determines whether the first or the second element of the list p is selected for further execution. In the second case the delay of the delaylist is released by the function "." and the recursive call starts. If $x \leqslant 1$ then the constant "1" is selected form the list p and operators in the delaylist are not executed at all.

To prove the program correctness we have to show the truth of the triple (3). Let us transform the triple to first-order logic formulas using the rule of forward tracing (1) based on the axioms for built-in functions. When execution of the function `fact` starts, the input argument `x` and the constant `1` form

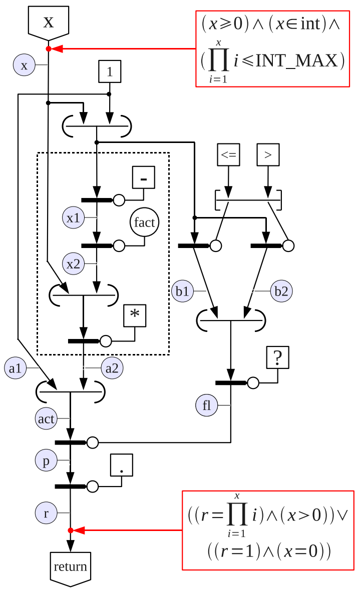

x
$(x \geqslant 0) \wedge (x \in \text{int}) \wedge (\prod_{i=1}^{x} i \leqslant \text{INT\_MAX})$
1
-
x1
fact
x2
*
<=
>
b1
b2
?
a1
a2
act
fl
p
.
r
$((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee ((r = 1) \wedge (x = 0))$
return

Fig. 2. The data-flow graph of the Hoare triple for function `fact`

the list `(x,1)`. Then functions “`<=`” and “`>`” are applied to this list. These are built-in functions with the same axioms that differ from each other only in the sign of the operation. For instance, axioms of the function “`<=`” are the following:

$$\boxed{\begin{array}{l}(p_1, p_2 \in \{int, float\}) \vee \\ (p_1, p_2 \in char) \vee \\ (p_1, p_2 \in bool)\end{array}} \; \texttt{(p1,p2):<=} \to r_1 \; \boxed{\begin{array}{l}(r_1 \in bool) \wedge \\ (r_1 = p_1 \leqslant p_2)\end{array}}, \qquad (4)$$

$$\boxed{\begin{array}{l}\neg((p_1, p_2 \in \{int, float\}) \vee \\ (p_1, p_2 \in char) \vee \\ (p_1, p_2 \in bool))\end{array}} \; \texttt{(p1,p2):<=} \to r_1 \; \boxed{\begin{array}{l}(r_1 \in error) \wedge \\ (r_1 = \texttt{BASEFUNCERROR})\end{array}}. \qquad (5)$$

Before applying the rule of forward tracing it is needed to choose only axioms, that satisfy condition A_2 in (1). The condition holds only for axiom (4) and never holds for axiom (5), so the axiom (5) is excluded as no execution could reach this path.

As the result of applying the forward tracing rule based on the axiom (4) to the triple (3) the following triple is obtained:

$$\boxed{\begin{array}{l}(x \in int) \wedge (x \geqslant 0) \wedge (\prod_{i=1}^{x} i \leqslant \texttt{INT\_MAX}) \wedge \\ (b_1 \in bool) \wedge (b_1 = (x \leqslant 1))\end{array}} \; \texttt{f1} \to r \; \boxed{\begin{array}{l}((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee \\ ((r = 1) \wedge (x = 0))\end{array}}, \qquad (6)$$

where `f1` corresponds to the code:

```
[( 1, { (x, (x,1):-:fact ):*  } ):(b1,(x,1):>):?]:.
```

After consecutive application of the forward tracing rule based on the axioms for built-in functions "`>`", "`?`", for the "datalist element selection function" and the function "`.`", we obtain the following triples:

$$\boxed{\begin{array}{l}(x \in int) \wedge (x \geqslant 0) \wedge (\prod_{i=1}^{x} i \leqslant \mathtt{INT\_MAX}) \wedge \\ ((x \leqslant 1) = true) \wedge ((x > 1) = false) \wedge (r = 1)\end{array}} \ \boxed{\begin{array}{l}((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee \\ ((r = 1) \wedge (x = 0))\end{array}}, \quad (7)$$

$$\boxed{\begin{array}{l}(x \in int) \wedge (x \geqslant 0) \wedge (\prod_{i=1}^{x} i \leqslant \mathtt{INT\_MAX}) \wedge \\ ((x \leqslant 1) = false) \wedge ((x > 1) = true)\end{array}} \ \mathtt{f2} \to r \ \boxed{\begin{array}{l}((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee \\ ((r = 1) \wedge (x = 0))\end{array}}, \quad (8)$$

where `f2` corresponds to the code:

```
(x, (x,1):-:fact ):*
```

The triple (7) has an "empty" program so according to the rule (2) it could be transformed into the following formula:

$$\Bigg((x \in int) \wedge (x \geqslant 0) \wedge (\prod_{i=1}^{x} i \leqslant \mathtt{INT\_MAX}) \wedge (x \leqslant 1) = true \wedge (x > 1) = false \wedge$$
$$\wedge (r = 1)\Bigg) \Rightarrow \Bigg(((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee ((r = 1) \wedge (x = 0))\Bigg) .$$

Obviously this formula is identically true, as $(x \geqslant 0)$ and $((x \leqslant 1) = true)$ implies $x \in \{0, 1\}$ and $r = 1$.

Now let us consider the triple (8). The function "`-`" is applied directly to the input argument x. By applying the forward tracing rule based on the axiom for function "`-`" to the triple (8) the following triple is obtained:

$$\boxed{\begin{array}{l}(x \in int) \wedge (x \geqslant 0) \wedge \\ (\prod_{i=1}^{x} i \leqslant \mathtt{INT\_MAX}) \wedge \\ ((x \leqslant 1) = false) \wedge \\ ((x > 1) = true) \wedge \\ (x_1 \in int) \wedge (x_1 = x - 1)\end{array}} \ \text{(x, x1:fact):*} \to r \ \boxed{\begin{array}{l}((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee \\ ((r = 1) \wedge (x = 0))\end{array}}. \quad (9)$$

Further the function `fact` is called recursively, so it is impossible to use the rule of forward tracing directly as it is still not proved that the function `fact` is correct.

Let the function argument "`x`" be called "the current argument" and "`(x,1):-`" be called "the argument of the recursive call" (it is denoted by x_1). The branch

a_1 of the data-flow graph has no recursive call, but the branch a_2 has one. So the conditions of recurring and leaving the recurrent function are $b_1 := (x \leqslant 1)$ and $b_2 := (x > 1)$, respectively (all identifiers are shown in the figure 2).

It is easy to show that the argument of the recursive call is correct as the condition A_2 in (1) for the recursive call of the function `fact` holds.

Let us define the bound function for the function `fact`:

$$f(x) = \begin{cases} 1, \ x = 0; \\ x, \ x > 0. \end{cases}$$

Let us prove the correctness of function `fact` inductively by the values of the bound function f.

The Basis. If the argument x satisfies the triple (3) precondition $(x \in int) \wedge (x \geqslant 0) \wedge \prod_{i=1}^{x} i \leqslant \texttt{INT\_MAX}$ and the condition of leaving the recurrent function $x \leqslant 1$ then $x = 0$ or $x = 1$. In this case $f(x) = f(0) = f(1) = 1$. It is also obvious that in this case the function `fact` finishes its execution without any recursive call and the return value is equal to 1 and satisfies the triple (3) postcondition. The formal check of this statement is equal to the formal proof of the Hoare triple (7), that was done earlier in the paper.

The Inductive Step. Let us take the argument x that satisfies the triple (3) precondition and the condition of recurring $x > 0$ for which the value of the bound function equals N: $f(x) = N$. Assume that the function is correct for all arguments for which the bound function values are less than N. Let us show that the value of the bound function for the argument of the recursive call is always less than N:

$$f(x_1) = f(x - 1) = f(N - 1) = N - 1 < N \ .$$

Then, according to the inductive assumption, the function `fact` triple for the argument of the recursive call is correct:

$$\boxed{\begin{array}{l} (x_1 = x - 1) \wedge \ (x_1 \in int) \wedge \\ (x_1 \geqslant 0) \wedge \ (\prod\limits_{i=1}^{x_1} i \leqslant \texttt{INT\_MAX} \ ) \end{array}} \ \texttt{x1:fact} \ \to x_2 \ \boxed{\begin{array}{l} ((x_2 = \prod\limits_{i=1}^{x_1} i) \wedge (x_1 > 0)) \vee \\ ((x_2 = 1) \wedge (x_1 = 0)) \end{array}} \ . \qquad (10)$$

The above triple could be used as the theorem in the program `fact` correctness proof with the help of the forward tracing rule, when the function `fact` is considered as nonrecursive. Then, by applying the forward tracing rule based on the theorem (10), to the triple (9) the following triple is obtained:

$$\boxed{\begin{array}{l} (x \in int) \wedge (x \geqslant 0) \wedge \\ (\prod\limits_{i=1}^{x} i \leqslant \texttt{INT\_MAX}) \wedge \\ ((x \leqslant 1) = false) \wedge \\ ((x > 1) = true) \wedge \\ (x_2 \in int) \wedge (x_2 = \prod\limits_{i=1}^{(x-1)} i) \end{array}} \ (\text{x, x2 }):^* \to r \ \boxed{\begin{array}{l} ((r = \prod\limits_{i=1}^{x} i) \wedge \ (x > 0)) \vee \\ ((r = 1) \wedge (x = 0)) \end{array}} \ . \qquad (11)$$

There is only multiplication "*" in the code part of the triple (11). Applying the forward tracing rule based on the axiom for this function we obtain the triple with the "empty" program:

$$\boxed{\begin{array}{l}(x \in int) \wedge (x > 1) \wedge (\prod\limits_{i=1}^{x} i \leqslant \texttt{INT\_MAX}) \wedge (x_2 \in int) \wedge \\ (x_2 = \prod\limits_{i=1}^{(x-1)} i) \wedge (x * x_2 \in int) \wedge (r = x * x_2)\end{array}} \quad \boxed{\begin{array}{l}((r = \prod\limits_{i=1}^{x} i) \wedge (x > 0)) \vee \\ ((r = 1) \wedge (x = 0))\end{array}} .$$

After using the rule (2) and simplification we obtain:

$$\left((x, x_2 \in int) \wedge (x > 1) \wedge (\prod_{i=1}^{x} i \leqslant \texttt{INT\_MAX}) \wedge (x_2 = \prod_{i=1}^{(x-1)} i) \wedge (r = x * x_2) \right) \rightarrow$$
$$\rightarrow \left(((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee ((r = 1) \wedge (x = 0)) \right) .$$

This formula is identically true, which means that the initial Hoare triple (3) is also true and this, in turn, implies the correctness of the program `fact`.

5 Conclusions

This paper describes an alternative approach to verification of parallel programs written in functional data-flow programming paradigm. A formal system, sufficient for proving the correctness of a program written in the Pifagor language, is considered. This method could be used as a base of a toolkit to support program correctness proving, since this method could be made automatic at many stages.

References

1. Nepomnyashiy, V.A., Ryakin, O.M.: Applied Methods for Programs Verification. Radio i svyaz, Moscow (1988)
2. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. CACM 10(12), 576–585 (1969)
3. Anureev, I.S., Maryasov, I.V., Nepomniaschy, V.A.: The Mixed Axiomatic Semantics Method for C-program Verification. MAIS 17(3), 5–28 (2010)
4. Udalova, U.V., Legalov, A.I., Sirotinina, N.U.: Debug and Verification of Function-Stream Parallel Programs. Journal of Siberian Federal University. Engineering and Technologies 4(2), 213–224 (2011)
5. Legalov, A.I.: The Functional Programming Language for Creating Architecture-Independent parallel Programs. Computational Technologies 10(1), 71–89 (2005)
6. Kropacheva, M.S.: Formalization of the Semantics for the Functional Data-Flow Parallel Language Pifagor. In: 12th Russian Scientific-Practical Conference "The Problems of Region Informatization", pp. 144–148. Siberian Federal University, Krasnoyarsk (2011)

Characterization and Understanding Machine-Specific Interconnects

Vitali Morozov*, Jiayuan Meng, Venkatram Vishwanath, Kalyan Kumaran, and Michael E. Papka

Argonne Leadership Computing Facility,
Argonne National Laboratory, Argonne, IL 60439, USA
{morozov,jmeng,venkat,kumaran,papka}@anl.gov

Abstract. The use of efficient communication patterns is becoming increasingly important to utilize high concurrency and achieve high throughput and low latency of modern high-performance supercomputers. Efficacy is not only dictated by an application communication pattern, but also driven by the interconnect properties, the node architecture, and the quality of runtime communication libraries. Different systems require different tradeoffs with respect to communication mechanisms, which can impact the choice of application implementations. We use the in-house MPI benchmark suite to study the communication behavior of the interconnects and guide the performance tuning of scientific applications. We report the results of our investigation of four supercomputer systems located at ALCF, and present lessons learned from our experience.

Keywords: interconnect, performance tuning, topology-aware, MPI.

1 Introduction

Current and future systems are becoming increasingly characterized by complex hierarchy and inherent parallelism both within the node and between the nodes. For a single node, we are witnessing diverse topologies to connect node components. We experience the use of rings (Blue Gene/Q, Xeon Phi), meshes (AMD, Tilera), or hierarchical topologies (Nvidia) with different latency and bandwidth capabilities. At a system level, high radix interconnects, such us toruses (Blue Gene, K Computer) and trees (Tianhe-1A) continue to dominate in the high-end systems. A common trend is that interconnects within a node are getting as complex and diverse as system interconnects. A critical challenge facing applications, runtime systems, and programming models is to fully exploit their hierarchies and specific features.

Our experience with scientific applications at the Argonne Leadership Computing Facility (ALCF) shows that efficacy of communication is critical to overall performance of an application. For example, a turbulence code DNS3D spends 40% of its execution time in communication; the computational fluid dynamics (CFD) applications, such as Nektar [1], typically spend 20% of time in

* Corresponding author.

communication. To reduce the communication overhead, one can apply techniques such as maximizing messaging rate, applying particular message orders, aggregating small messages, overlapping communication with computation, etc. However, different systems demonstrate different responses to these techniques. The tradeoff involves several hardware and runtime factors, such as communication protocols, interconnect latency and bandwidth, node concurrency, threading strategies, number of available links, topology of interconnect, as well as specific interconnect and implementation features, such as availability of floating point operations in routers, remote memory accesses, and overheads in MPI libraries.

The dynamic tradeoffs differ across applications or even application phases, and it is often unclear how to choose the best optimization techniques. Empirically tuning each possiblity can be time consuming, if at all feasible. Furthermore, it remains unknown how much performance gains can be achieved with further tuning. The intrinsic properties of the interconnect may even prevent application from a particular optimization. For example, adaptive mesh refinement (AMR) applications usually maintain a list of neighboring elements; the efficient data exchange between these elements strongly depends on the number of outstanding requests that each node can effectively support. It is therefore necessary to understand the low-level behavior of the system interconnect and apply such knowledge to application implementations in a more educated way. The vendors do not provide such information.

There is a rich set of existing MPI benchmarks available to test various aspects of system interconnects. SPEC MPI 2007 [2] and NAS Parallel Benchmarks [8] provide a set of mini applications from various scientific domains. Each of them has differently exercised communication aspects of unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grid patterns. The performance of specific MPI primitives cannot be extracted out separately from the results of the entire run to better understand the performance of the application's MPI scenario. The scalability of the suites is limited to thousands of cores, which is not applicable to a large-scale leadership-class facilities. The mini application benchmarks are complimentary to our effort wherein we are trying to understand the behavior of the system at a finer level of detail with specific communication scenario.

The Ohio State University MPI Benchmarks [5] consist of point-to-point (p2p) communication benchmarks, primarily focused on latency and bandwidth measurements between a pair of nodes. It has also incorporated support for one-sided communications for CUDA devices. Similarly, both Intel MPI [3] and SKaMPI [6] provide a set of micro-benchmarks with elementary MPI operations; they measure the latency of isolated communication operations, but do not study the performance impact when communication operations are issued in different order, rate, parallelism, and under different interconnect partitions and topologies. Phloem MPI benchmark suite [4] provides a collection of elementary MPI tests, and we were intensively discussing with the Phloem developers the applicability of their suite to ALCF workloads. It was agreed that Phloem does not focus on studying the effects of systems topologies, efficacy of using multiple communicators and sub-communicators, impact of the operation orders, and computation-communication interaction, among others.

The performance of tuned communication subsystems with the use of parallel communication models was previously reported in [9–11]. They primarily focused on giving recommendations to engineers on ways to improve the next-generation interconnects, and the results cannot directly be used by application developers.

The ALCF is home to several leadership-class machines; it also accommodates a representative set of production scientific applications. Based on our experience at the ALCF, we have generalized a number of representative communication schemes and developed an ALCF MPI benchmark suite [7] - a set of micro benchmarks and mini apps to measure the interconnect hardware response directly and gain deeper understanding on communication internals. Key features of our benchmark suite include studying the capabilities of node messaging layer to saturate the interconnect, understanding the tradeoffs between single link versus multiple links performances, measuring latency and bandwidth of various operations subject to sub-communicators constrains, studying nearest neighbor and halo exchange capabilities, tuning the efficacy of overlapping computation with communication, and others. Some MPI benchmarks can be adopted to the scenarios we exercise by elaborate placement of MPI processes to a specific core by using mapping files. While appropriate at a small number of processes and simple interconnect topologies, this will quickly become intractable for large systems with hundreds of thousands of processes and complex system topologies; and therefore, the intrinsic topology awareness must be incorporated in the benchmark itself.

The rest of the paper is organized as follows. We introduce the systems used for our evaluation in Section 2. Results of the experiments are presented and discussed in Section 3. Conclusions, lessons learned, and discussion of future plans are covered in Section 4.

2 Evaluation Environments

We characterize the interconnect properties of four ALCF systems: an IBM Blue Gene/P (BG/P) supercomputer (`Intrepid`), IBM Blue Gene/Q (BG/Q) supercomputer (`Mira`), and two heterogeneous CPU-GPU clusters, one with a 10Gb Ethernet (`Eureka`) and the other with an Infiniband-based (`Magellan`) interconnects. We use IBM XL compiler with default communication libraries on Blue Gene supercomputers, and GNU compiler with MPICH on clusters.

`Intrepid` is a 163,840-core BG/P supercomputer with a peak performance of 557 TFlops, 80 TB memory, and 640 I/O nodes [12]. A node is a quad-core, 32-bit, 850 MHz IBM Power PC 450 with 2 GB of DDR2 memory, a coherent cache hierarchy with L3 cache shared among cores, and can run up to 4 MPI processes simultaneously. Inter- and intra-node communication is handled by a messaging unit that connects to the L3 cache with a data bus and a control bus. This unit is shared by all MPI processes on the same node. If a message to be communicated resides in main memory, it needs to be first loaded into the L3 cache before going off-node. The system is built from individual racks; each rack contains 1,024 compute nodes, for a total of 4,096 cores per rack. At the ALCF, 64 nodes are grouped into a *pset*, the smallest partition, that can be allocated to a user. Compute nodes are connected over three interconnects. The 3-D torus is used for p2p communications over 12 425-MB/s links which enable bidirectional connections to six nearest neighbors. The tree-structured collective network is used for collective and I/O communications. It supports integer and double precision floating point reductions. The bandwidth per link is 850 MB/s, and the hardware latency is 1.3 μs per tree traversal. The global barrier and interrupt network is used for synchronization.

`Mira` is a 786,432-core BG/Q supercomputer with a peak performance of more than 10 PFlops, 786 TB memory and 384 I/O nodes [11]. A new compute chip design combines the 17 Power A2 cores, caches, a crossbar, and a messaging unit. Each core is a 64-bit Power ISA with 4 hardware threads, 4-way SIMD FPU vector unit, running at 1600 MHz. 16 cores are available to a user and can run up to 64 MPI processes simultaneously. One core is used for handling OS interrupts and other system services. A node contains one compute chip, 16 GB of DDR3 memory, and 10 bidirectional communication links with a peak total bandwidth of 40 GB/s. BG/Q uses a single 5-D torus interconnect to handle p2p, collective, I/O, and synchronization operations. As its predecessor, BG/Q is built from individual 1,024-node racks for a total of 16,384 cores per rack. At the ALCF, the smallest partition is 128 nodes.

`Eureka` is a 100-node compute cluster, which is primarily used for data preparation, data analysis, and visualization. Each node has a dual-socket quad-core Intel Xeon E5405 server with two NVIDIA Quadro FX5600 GPUs. The nodes are connected using a 10-Gb-Ethernet Myrinet switch configured as a 5-stage CLOS network.

`Magellan` is 100-node compute cluster, which is used for data analysis and visualization of data obtained on `Mira`. Each node has a dual-socket eight-core AMD Opteron 6128 processors with two Nvidia Fermi M2070 GPU cards. The nodes are connected via a QDR Infiniband interconnect.

3 Experiments

Depending on communication operation characteristics, the MPI library may implicitly handle it using different protocols and invoking different hardware units. For example, the MPI library often uses one protocol for small messages and a different protocol for large messages. On the other hand, the applications can also be using different strategies with regard to which MPI function to use, how to order messaging operations, which communicators to construct, etc. Because a system often provides various hardware and software options in system communication layer, the choice of the winning strategy is often not obvious for application developers. By running the set of benchmarks, we have identified a number of interesting facts and concluded several heuristics for programming efficient communications.

3.1 Messaging Rate

The messaging rate benchmark measures the capabilities of a single node interconnect subsystem to drive all available links simultaneously at full speed. It reports the number of messages, in millions, that can be communicated to and from a node within a second (mmps). A typical user scenario, which exercises a messaging rate communication pattern, is common for high-order spectral element CFD codes, such as Nektar. In these codes, the domain partitioning dictates the usually complex irregular distribution of spectral elements over MPI processes. The virtually adjacent processes, that share the vertices and faces of the same element, may be located far away from each other. Each process may therefore handle tens to hundreds of "neighbor" processes exchanging messages all at the same time. The message sizes can range from hundreds of bytes to tens of kilobytes.

The rate is measured on a given reference node by aggregating the number of messages sent by all MPI tasks on this node. To maximize the rate, each task on a reference node is communicating with one task per available link on all neighbor target nodes. Therefore, if one task is placed on a BG/P reference node, it will communicate to six neighbor nodes – one per each torus direction. Similarly, two tasks on a reference node will communicate to 12 tasks on the same neighbor nodes. A BG/P node can handle up to four MPI tasks, one per each core. On the BG/Q node, each node has nine nearest neighbors, because the "E" direction of the torus has always size 2. Each core can handle one, two, or four MPI tasks; however, the tasks start sharing the resources of the core. Both `Eureka` and `Magellan` clusters are using a single bus to connect to an interconnect, and therefore they only run 1 task on a reference node.

According to our measurements, a single MPI process is unable to saturate the capabilities of Blue Gene communication subsystem. On `Intrepid`, the rate linearly increasing from 0.63 for 1 task, to 1.25 for 2 tasks, to 2.41 for 4 tasks. This is expected for a well-balanced system where all compute cores should be able to simultaneously communicate without performance degradation from the messaging subsystem. On `Mira`, 1 task demonstrated a rate of 0.90, 8 tasks – 6.45, 16 tasks – 11.27, and 32 tasks saturate the messaging unit with 15.87 mmps. The Power A2 core can issue two instructions per clock cycle only if two processes are running on the core - one instruction is taken from each process. Placing more processes per core has no advantages for the throughput and, in fact, the processes start competing for the shared caches. Indeed, 64 tasks have demonstrated a slight degradation 10.25 mmps from the best measured result.

The messaging rate on Blue Gene systems is very high compared to the one obtained on `Eureka`, which yields only 0.08 mmps. Ethernet typically provides relatively high communication latency, compared to both low-latency Infiniband and proprietary interconnects. Indeed, `Magellan`'s messaging rate is much higher and reaches 0.90 mmps, placing itself between BG/P and BG/Q.

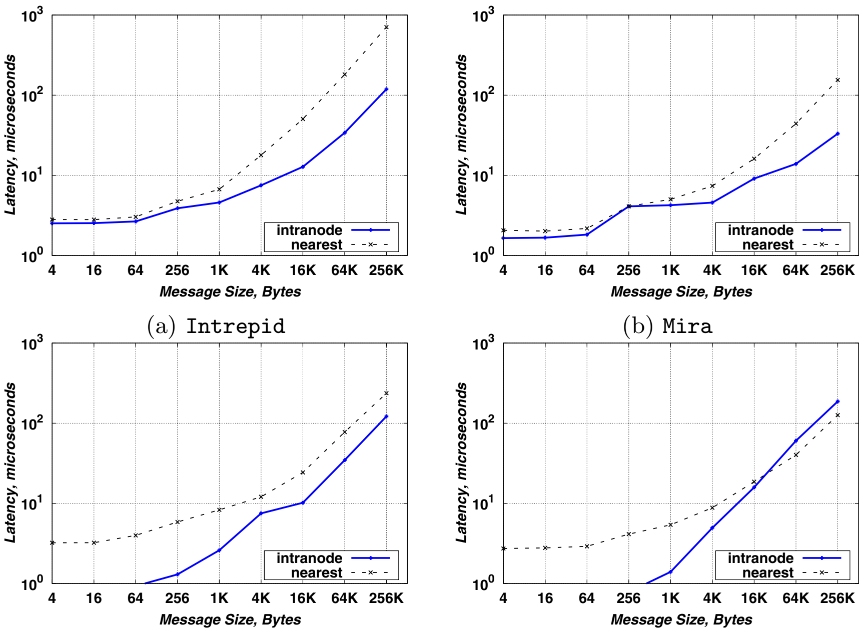
We conclude that Blue Gene messaging units can efficiently host continuous communication requests from all cores simultaneously. For an application where messaging rate is critical for performance, running at least one MPI task per core will yield the optimal performance. Users should optimize the placement of ranks to use more links at a time.

3.2 Point-to-Point Communication Latency

Point-to-point communication latency is obtained by using a ping-pong benchmark, which measures the time for executing a pair of blocking MPI send/receive of a message of specified length between two tasks, and then calculates the one-way latency, dividing the round trip latency by two. Placement of participating tasks provides three distinct variants of execution which require separate consideration and analysis. For `intranode`, the communicating tasks are sharing the same node, and the latency is determined by the properties of node memory hierarchy and the capabilities of messaging software stack to efficiently utilize them. Although rare in practice, an optimized stack may detect that the two participating tasks reside on the same node, and take advantage from direct *memcpy* routines. For `nearest` neighbor, the communicating tasks are placed into two adjacent nodes, and the latency is determined by link latency, message size, and software overhead, such as the effectiveness of message size-dependent protocols and message startup time. The benchmark helps to determine the slowest path in the messaging layer. For `farthest` pair of tasks, the tasks are placed to the nodes with the longest path in between, and the latency, additionally to the factors above, depends on the properties of the partition size and the routing protocol. An effective protocol may potentially reduce collisions and involve more links to a transfer, and thus increase available link bandwidth.

Figure 1 summarizes the results for `intranode` and `nearest` benchmarks. For small messages, both `Intrepid` and `Mira` show little latency difference, reflecting that the hardware overhead is only a small fraction of the total communication time. This explains the controversy in various discussions for Blue Gene mapping strategies, when most communicating tasks are placed to different nodes [13]. For large messages, the determining factor is eventually a ratio of memory bandwidth versus single link bandwidth. In all cases, `Mira` has 1.5 to 5 times lower latency due to hardware improvements. On regular clusters, the tasks on the same node are communicating almost an order of magnitude faster than the tasks on different nodes.

Figure 2 presents the results of the `farthest` p2p benchmark for `Intrepid` and `Mira`. We use 32,768 nodes on both machines; however, the diameter of the BG/Q partition is much smaller than that of the BG/P. We also include the results of `nearest` benchmark for reference. On both supercomputers, the `nearest` benchmark incurs similar latency to the farthest latency, reflecting that the overhead of multiple routers is negligible. This indicates that it is more worthwhile to locate the ranks in a way that utilizes more links at a time, than to optimize the aggregate distance among communicating ranks. This observation continues to be true on the BG/Q even for very large partitions up to the entire machine.



(a) `Intrepid` (b) `Mira`

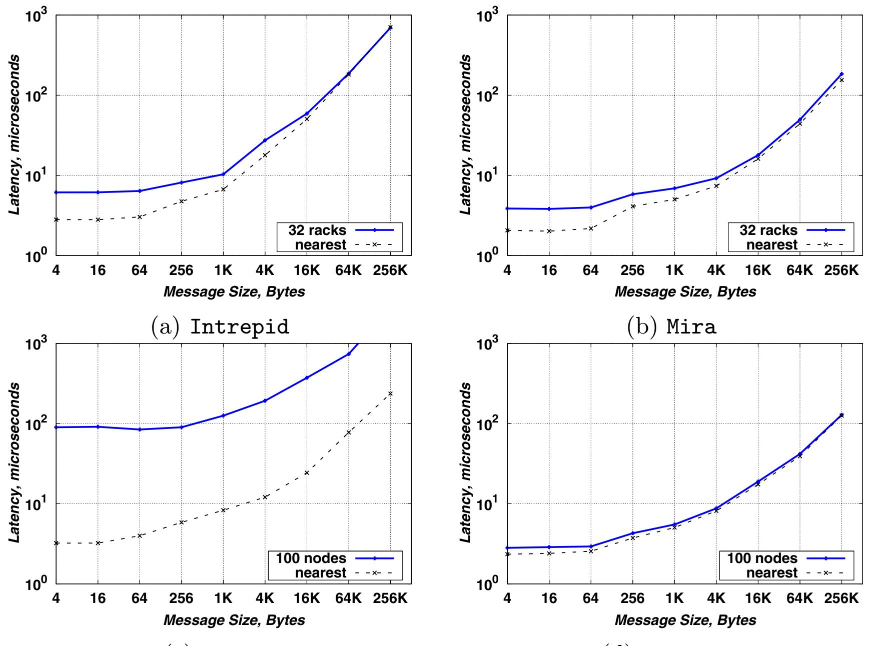
(c) `Eureka` (d) `Magellan`

Fig. 1. Latency of `intranode` and `nearest` p2p communications

3.3 Aggregate Node Bandwidth

This benchmark measures the `aggregate` bandwidth for incoming and outgoing communications between the tasks placed on a reference node and the corresponding tasks located on all its nearest neighbors. The benchmark incorporates the same topology awareness and messaging logic, as we discussed in Section 3.1, except that this benchmark measures the capabilities of interconnect to drive all available links simultaneously at full speed. By changing the message size, the benchmark also helps to determine the largest data block that should be used by applications before interconnect becomes a bottleneck. We present the results for 512 KB messages.

From the row link bandwidth we have determined that the theoretical aggregate peak bandwidths for `Intrepid` around 4.53 GB/s, for `Mira` – 36 GB/s, for `Eureka` – 2.50 GB/s, and for `Magellan` – 3.20 GB/s. According to the measurements, `Intrepid`'s aggregate node bandwidth is 4.28 GB/s on 1 MPI task and 4.38 GB/s with 2 and 4 MPI tasks, 97% of peak. `Mira` has shown 31.49 GB/s with 1 MPI task, 35.59 GB/s for 2 to 16 MPI tasks, and 32.07 GB/s for 64 MPI tasks, 99% in best and 87% in worst case. We have already learned from other benchmarks that 1 task on a node cannot drive the messaging unit at full speed. `Magellan`'s InfiniBand has delivered 2.70 GB/s, `Eureka` shows 2.44 GB/s.



(a) Intrepid (b) Mira (c) Eureka (d) Magellan

Fig. 2. Latency of `farthest` p2p communications

3.4 Bisection Bandwidth

Bisection bandwidth of a partition is often the limiting performance factor for `all-to-all`-type communication patterns which occur in applications, such as 3D FFT, parallel dense linear algebra, or turbulent CFD. It is usually defined as the minimum delivered bi-directional bandwidth over all possible bisections of a partition. Depending on the topology of a partition, such a bisection can be constructed without trial of all bisections. For torus topologies, the bisection is defined as a cross cut over the longest dimension of the partition. For tree-like topologies, the cross cut is generally made over the top-level parent of a tree.

Table 1. Bisection bandwidth, in TB/s

	`Mira`			`Intrepid`		
	Peak	Measured	%peak	Peak	Measured	%peak
512 nodes	1.02	0.91	89	0.01	0.01	84
8 racks	4.10	3.64	89	0.44	0.38	88
32 racks	16.38	14.55	89	1.74	1.54	88
48 racks	24.58	21.83	89			

We measure the bisection bandwidth for partitions sized from 512 nodes to 48 racks. Table 1 shows results of bisection measurements for various partition sizes on `Intrepid` and `Mira`. The theoretical peak bandwidth was calculated based on the number of available links between the bisections and the row speed of each link. Both BG/P and BG/Q are capable to deliver near 90% of available row bandwidth for all the partitions – an overhead of about 10% is due to control information, which is stored with each packet of the message. We conclude, that bandwidth intensive applications with `all-to-all`-type will favor the properties of the BG/Q interconnect, high effective bandwidth of sub partitions, as well as high effective bandwidth, available to an application.

3.5 Collectives Latency

We study the latency of collectives by benchmarking the three most frequently used operations, `broadcast`, `reduction`, and `barrier`, by using their correspondent MPI functions. Due to large variability of benchmark cases and parameters, we only discuss the results for those, which reflect, in our opinion, the most distinct features of the ALCF supercomputing environment.

Comparing `broadcast` and `reduction` latencies on 1 rack of `Intrepid` versus 1 rack of `Mira` on `World` communicator (1 MPI task is used for all benchmarks), we confirmed that the BG/Q interconnect delivers better performance than the BG/P interconnect. On BG/Q, the latency for `broadcast` dropped down 47%, from 4.58 μs to 3.12 μs, the latency for `reduction` dropped down 64%, from 5.53% μs to 3.38 μs. The difference becomes more pronounced for larger partitions, and this is due to the very small diameter of the 5-D torus and much higher single link raw bandwidth. Contrary, `barrier` is about two times slower, 2.24 μs on BG/Q versus 1.19 μs on BG/P. The use of dedicated global barrier network clearly helps to reduce synchronization time on BG/P; however, this fact may hardly be accounted as a weak point of low-latency BG/Q interconnect.

The choice of a communicator may greatly affect the latency of collectives, because the communication library usually uses different routing and optimization strategies. We found that the BG/P communication library uses fast collective network only if an entire `World` communicator is used; in other cases, the library chooses to use a torus network. Performance degradation can be very significant. For example, torus implementation of `broadcast` is about 25% slower, 5.69 μs; `reduction` is about 7 times slower, 37.50 μs; and `barrier` is 18 times slower, 21.8 μs. The BG/P interconnect is designed for fast communication across entire partition and does not favor the communication over sub-partitions. Unfortunately, most applications are unaware of such artificial limitation and widely use the hierarchy of sub-communicators, such as often occurring, for example, in multigrid solvers. The BG/Q interconnect does not show such a limitation, at least for rectangular sub-communicators.

Next, we evaluate the impact of scaling the number of MPI ranks per node on the latency of collective operations. Table 2 depicts the latency, in μs, on 1 rack of `Intrepid` and `Mira` for various numbers of MPI tasks per node. On `Mira`, the latency gradually increases for larger number of tasks (collective on a node is performed in software and, therefore, the increase is noticeable), and it jumps near 3 times for 64 tasks. The low-level communication library relies on system "communication thread," which progresses communication operation in the background of the main thread. With 64 tasks per node, this system thread is unavailable, and the progress must be made by the main MPI thread, which leads to constant interrupts and therefore, performance degradation. We conclude, that even though the "64 MPI tasks per node" mode is available to a user; it should be used with caution, because it is exposed to increased resource contention.

Table 2. Latency, μs, for multiple number of tasks in a node on 1 rack (1,024 nodes)

	`BG/Q`				`BG/P`		
Number of tasks per node	1	4	16	64	1	2	4
`barrier`	2.25	3.56	3.73	9.21	1.19	2.29	3.06
`broadcast` 16 bytes	4.01	4.27	4.51	8.70	4.58	7.31	7.87
`broadcast` 8192 bytes	9.30	9.49	9.61	17.2	20.1	26.8	27.5
`reduction` 1 double	4.34	4.95	6.15	14.5	5.52	6.60	7.40
`reduction` 256 doubles	5.85	8.49	11.4	29.8	28.6	37.9	43.3

As shown in Table 3, BG/Q continues to demonstrate high scalability of collective operations in a multi-rack configuration. Our previous findings are entirely confirmed at scale. The latency of a collective does not depend on the size of the partition. Placement of MPI tasks on a node has a bigger impact on the latency, and running 64 MPI tasks per node can be practical for some use cases. Performance of a collective on a rectangular sub-communicator is as optimized as that on an entire partition.

3.6 Halo Exchange Latency

The 3-D halo exchange communication pattern is arguably the most commonly used data exchange in scientific applications. Each MPI task requires data produced on the ranks with neighboring coordinates (i.e., halo), and therefore has to exchange halo data with its neighbors. We use the `halo exchange` benchmark with four different halo exchange implementation strategies, discussed in [7] in detail. In the following, we summarize the key features of the implementation.

`Halo exchange` benchmark investigates several communication strategies denoted: `Sendrecv`, `Isend-recv`, `Isend-Irecv`, and `AllAtOnce`. In the `Sendrecv` strategy, each task communicates along dimensions in order; within each dimension, the tasks issue two blocking `MPI_Sendrecv` calls to communicate to the peers in forward and in backward directions. In the `Isend-recv` strategy, a task first issues a non-blocking `Isend` request, then synchronously `receives` the data from its peer, not waiting for the send to complete. A `barrier` ensures that communication in one direction finishes for *all* ranks before moving on to the next direction. This strategy typically leads to a flood of unexpected messages, which

Table 3. Latency, in microseconds, of multi-rack BG/Q collectives for various ranks per node (RPN)

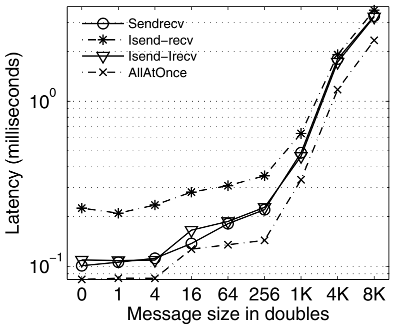
	`World` communicator			`Half-World` sub-communicator		
RPN	8 racks	32 racks	48 racks	8 racks	32 racks	48 racks
	`barrier`					
1	3.92	4.84	5.03	3.84	4.41	4.64
16	5.42	6.93	7.54	5.33	5.96	6.19
64	11.02	11.88	14.46	10.74	11.73	11.79
	`broadcast` 16 bytes					
1	4.88	5.84	6.05	4.78	5.44	5.64
16	6.26	7.25	7.48	6.20	6.84	7.06
64	10.74	11.52	11.90	10.68	11.28	11.57
	`broadcast` 8192 bytes					
1	9.85	10.80	11.02	9.73	10.39	10.59
16	11.41	12.48	12.68	11.20	11.98	12.13
64	18.66	19.83	19.98	18.56	19.21	19.61
	`reduction` 1 `double`					
1	5.32	5.32	6.55	5.21	6.04	6.13
16	7.95	8.96	9.15	7.85	8.53	8.79
64	16.62	17.71	17.83	16.74	17.38	17.56
	`reduction` 256 `doubles`					
1	6.69	7.69	7.92	6.58	7.27	7.50
16	10.84	11.85	12.06	10.84	11.57	11.75
64	21.84	22.94	22.99	21.75	22.59	22.51

can degrade performance; however, this is a valid strategy observed in production ALCF workloads. The `Isend-Irecv` strategy relaxes synchronization for receives – a task posts non-blocking `send` and `receive`, completes them with `waitall`, and then starts next direction. Finally, the `AllAtOnce` strategy posts six non-blocking `sends` and `receives` for all dimensions simultaneously, after which `waitall` ensures that all outstanding requests are complete. All benchmarks are using the communicator returned by `cart_create` call with default task placement.

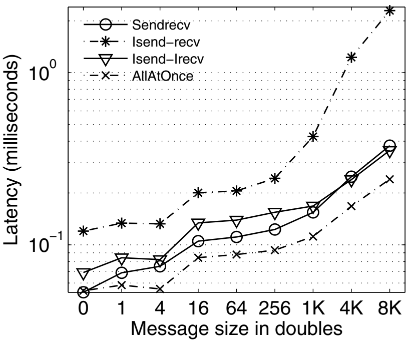
Performance of halo exchange pattern is sensitive to load imbalance. When tasks are running unevenly, some nodes start sending messages to the peers before matching `receive` is posted. The low-level communication library may either buffer the message on the sending side, causing extra overhead and increasing the chance of conflicts in messaging unit, or buffer the message on the receiving side, putting the result into the queue of *unexpected messages*. To study the system response, the benchmark puts a "sleep time" (delay) before halo exchanges to mimic the imbalance. The reported time is "true" communication time, where "sleep time" is excluded.

Both `Intrepid` and `Mira` exhibit similar response to different strategies; therefore, we only present the `Mira` results on Figure 3a. For messages close to 64-256 doubles (512 to 2,048 bytes), the effects of communication pattern are becoming more visible. Expectedly, the less synchronous `AllAtOnce` strategy is the fastest, and `Isend-recv` is the slowest. Figure 3b shows the results for imbalanced halo exchange overhead, which is the total amount of halo exchange time minus the specified amount of delay (a 100 ms in our case). A small imbalance in tasks helps the processes hide the communication progress over the compute part, and we observed that all communication strategies, except `Isend-recv`, benefit from such imbalance. `Isend-recv` performs the worst because the blocking `receive` inhibits a task from processing other messages, therefore messages sent by the previous non-blocking `send` are likely to reach the receiver before the corresponding `receive` is posted, resulting in extra overhead to manage unexpected messages.

We further compare the halo exchange latency by varying the number of MPI tasks per node. As Figure 4a shows, the BG/Q messaging unit is not entirely saturated. However, 64 MPI ranks per node incurs significantly more overhead than fewer ranks per node, which is primarily due to the inability of using a dedicated communication thread. Small messages are not able to saturate the available bandwidth; therefore, the latencies are comparable. For 1,024 doubles

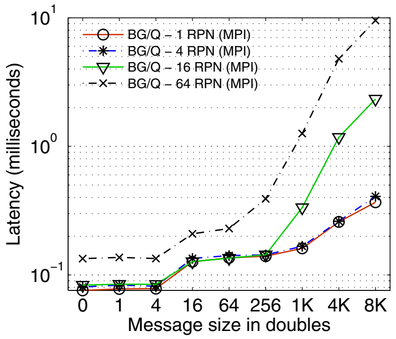


(a) `Halo exchange only`

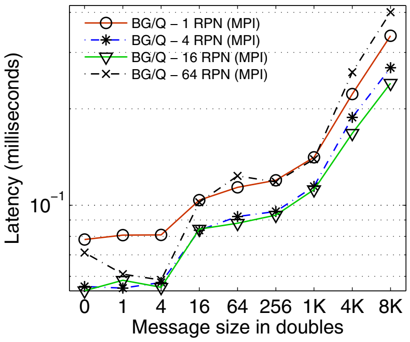


(b) `A 100 ms delay before exchange`

Fig. 3. Various halo exchange implementations with 16 tasks per node on `Mira`



(a) `Halo exchange only`



(b) `A 100 ms delay before exchange`

Fig. 4. `AllAtOnce` implementation at various number of tasks per node on `Mira` and larger, the messaging unit is saturated showing performance degradation when more tasks are placed on a node. When delay is added to mimic computation before halo exchanges, the MPI ranks on the same node are less likely to communicate at the same time, therefore the above degradation becomes negligible, as shown in Figure 4b.

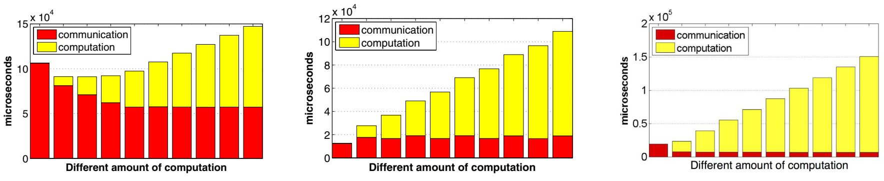
According to the benchmark study, our general recommendations for programming halo exchange applications on Blue Gene systems are very generic and entirely expected: a) do not synchronize processes without a reason; and b) do not predefine any order of communications. The Blue Gene interconnect is generally very fast and predefined ordering of send and receive requests does not bring benefits, but oppositely, degrades communication performance.

3.7 Overlapping Computation with Communication

We have discussed in a previous subchapter that the `AllAtOnce` strategy is the most efficient way to perform halo exchange. Some halo-exchange-based applications attempt to hide communication overhead by computing on local data in the background of communication. The `overlap` benchmark mimics this behavior. Each task posts six non-blocking `sends` and `receives`, sleeps for a specified delay time to imitate the processing in the background of communication, and finally posts `waitall` to complete the operation. One MPI task per core is used in all measurements.

The *overlap efficacy* is defined as the communication latency without delay divided by the halo exchange latency without overlapping. It is determined by both hardware and software capabilities. In some cases, CPU is involved in communication progress and compete for compute cycles. In such circumstances, the performance of the overlap strategy may get worse than the performance of independent halo exchange and correspondent computation phase.

Figure 5a demonstrates the `overlap` latency on `Intrepid`. Overlapping computation with communication can reduce communication overhead by about 40% on BG/P and 65% on BG/Q. However, as Figure 5b shows, the interrupt mode must be used to receive messages instead of default polling mechanism. Polling messages consumes CPU cycles, contends for resources with the computation, and may even lengthen the communication time. As also follows from Figure 5a, adding a little computation may slightly reduce the combined com-



(a) BG/P, Pooling (b) BG/P, Interrupt (c) BG/Q

Fig. 5. `Overlap` latency with 10 ms delay increments from 0 to 90 ms. Latency reduction reflects the efficacy of the overlap. On BG/P, interrupt mode is required. On BG/Q, default pooling is effective.

munication and computation time. We have identified that this happens when a slight imbalance in tasks puts less contention on the messaging unit by spreading its use more evenly over time.

4 Conclusion and Future Work

We have presented the results of our in-house ALCF MPI Benchmarks Suite to investigate the interconnect capabilities of four ALCF supercomputers. The main effort was put into studying the interaction of various factors that can impact the performance of the communication subsystem, such as the placement of processes in a node, the use of topology-awareness, the influence of load imbalance to latencies of elementary operations, as well as the use of low-level interconnect capabilities for typical halo exchange operations with and without computational overlap. Our findings for the IBM Blue Gene interconnect are:

- A node can effectively adopt multiple number of MPI processes and handle communication requests without conflicts in the messaging layer. Efficacy of processing small messages can be improved by increasing the concurrency of outstanding requests. Efficacy of processing large messages is limited by the off-node capabilities and aggregate bandwidth of the available links.
- Partitioning is one of the features of the Blue Gene interconnect. The peak aggregate and bisection bandwidth, as well as the best latencies for collectives are achievable if full partition with wrapped-up links is used. The use of smaller sub-partitions may or may not degrade performance. On BG/Q, the latency of collective operations on rectangular sub-communicators does not increase and is almost independent on the size of the partition.
- High concurrency is an important feature of the Blue Gene interconnect. Efficient halo exchange strategy should avoid synchronization of any kind and post all communication requests at once. Furthermore, a little load imbalance can help to reduce communication latency due to less likelihood of contention. Subject to certain restrictions, the overlapping communication with computation provides even more opportunities for interconnect utilization.

The lessons learned are helping us to port scientific applications from other platforms to an ALCF BG/Q supercomputer and improve the efficiency of their communication strategies. We have shown that the regular clusters demonstrate distinct communication behavior, and developers should adapt their applications in different ways to achieve high efficacy.

We believe that the ALCF MPI Benchmark Suite can guide system architects and application developers to design better hardware and to improve their applications. The benchmark was recently made open-source and released to the general public. As part of the next procurement cycle, our future directions are to continue working with supercomputer vendors and application developers in adopting the needs of pre-exascale era. We are looking at the ways to adopt legacy applications to an expected unprecedented node-level concurrency, the advent of millions of MPI processes in a single job, as well as interactions of emerging programming models with existing MPI-OpenMP runtimes. We will also be expanding the benchmark with other communication patterns as we find them in ALCF applications.

Acknowledgments. The research used resources of the ALCF at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. The authors would like to thank Bob Walkup and Daniel Faraj from IBM, and Kevin Harms from the ALCF for their help in this study. They are also grateful to the ALCF applications and operations staff for their assistance.

References

1. Grinberg, L., Morozov, V., Fedosov, D., Insley, J.A., Papka, M.E., Kumaran, K., Karniadakis, G.E.: A New Computational Paradigm in Multiscale Simulations with Applications to Brain Blood Flow. Gordon Bell Honorable Mention. In: Int. Conf. for High Performance Computing, Networking, Storage, and Analysis, Seattle, WA, November 12-18 (2011)
2. Müller, M.S., van Waveren, M., Lieberman, R., Whitney, B., Saito, H., Kumaran, K., Baron, J., Brantley, W.C., Parrott, C., Elken, T., Feng, H., Ponder, C.: SPEC MPI2007 – an Application Benchmark Suite for Parallel Systems using MPI. J. Concurrency and Computation: Practice and Experience – Int. Supercomputing Conf. 22(2), 191–205 (2010)
3. Intel MPI Benchmarks 3.2.3. Intel Corporation, `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`
4. Phloem MPI Benchmarks. Lawrence Livermore National Laboratory Technical report LLNL-MI-400479, `https://asc.llnl.gov/sequoia/benchmarks/`
5. The Ohio State University MPI Benchmarks. The Ohio State University, `http://mvapich.cse.ohio-state.edu/benchmarks/`
6. Augustin, W., Worsch, T.: The SKaMPI 5 Manual. Technical report (2008), `http://liinwww.ira.uka.de/~skampi/`
7. Morozov, V., Meng, J., Vishwanath, V., Hammond, J., Kumaran, K., Papka, M.: ALCF MPI Benchmarks: Understanding Machine-Specific Communication Behavior. In: Fifth Int. Workshop on Parallel Programming Models and Systems Software for High-End Computing, Pittsburgh, MA, September 10 (2012)
8. NAS Parallel Benchmarks. NASA Advanced Supercomputing Division, `http://www.nas.nasa.gov/publications/npb.html`
9. Underwood, K.D., Brightwell, R.: The Impact of MPI Queue Usage on Message Latency. In: Proc. Int. Conf. on Parallel Processing, vol. 1, pp. 152–160 (2004)
10. Brodsky, A., Pedersen, J.B., Wagner, A.S.: On the Complexity of Buffer Allocation in Message Passing Systems. J. Parallel Distrib. Comput. 65(6), 692–713 (2005)
11. Chen, D., Eisley, N.A., Heidelberger, P., Senger, R.M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D.L., Steinmacher-Burow, B., Parker, J.J.: The IBM Blue Gene/Q Interconnection Network and Message Unit. In: Int. Conf. on High Performance Computing, Networking, Storage and Analysis, Seattle, WA, November 12-18 (2011)
12. Overview of the IBM Blue Gene/P project. IBM J. Res. Dev. 52(1/2), 199–220 (2008)
13. Hoefler, T., Snir, M.: Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In: Proc. Int. Conf. on Supercomputing, Tucson, Arizona, pp. 78–84 (2011)

Multi-core Implementation of Decomposition-Based Packet Classification Algorithms[*]

Shijie Zhou, Yun R. Qu, and Viktor K. Prasanna

Ming Hsieh Department of Electrical Engineering, University of Southern California
Los Angeles, CA 90089, U.S.A.
{shijiezh,yunqu,prasanna}@usc.edu

Abstract. Multi-field packet classification is a network kernel function where packets are classified based on a set of predefined rules. Many algorithms and hardware architectures have been proposed to accelerate packet classification. Among them, decomposition-based classification approaches are of major interest to the research community because of the parallel search in each packet header field. This paper presents four decomposition-based approaches on multi-core processors. We search in parallel for all the fields using linear search or range-tree search; we store the partial results in a linked list or a bit vector. The partial results are merged to produce the final packet header match. We evaluate the performance with respect to latency and throughput varying the rule set size (1K ~ 64K) and the number of threads per core (1 ~ 12). Experimental results show that our approaches can achieve 128 ns processing latency per packet and 11.5 Gbps overall throughput on state-of-the-art 16-core platforms.

1 Introduction

Internet routers perform *packet classification* on incoming packets for various network services such as network security and Quality of Service (QoS) routing. All the incoming packets need to be examined against predefined rules in the router; packets are filtered out for security reasons or forwarded to specific ports during this process. Moreover, the emerging Software Defined Networking (SDN) requires *OpenFlow table lookup* [1], [2], which is similar as the classic multi-field packet classification mechanism. All these requirements make packet classification a kernel function for Internet.

Many hardware and software-based approaches have been proposed to enhance the performance of packet classification. One of the most popular methods for packet classification is to use Ternary Content Addressable Memories (TCAMs) [3]. TCAMs are not scalable and require a lot of power [4]. Recent work has explored the use of Field-Programmable Gate Arrays (FPGAs) [5]. These designs can achieve very high throughput for moderate-size rule set, but they also suffer long processing latency when external memory has to be used for large rule sets.

Use of software accelerators and virtual machines for classification is a new trend [6]. However, both the growing size of the rule set and the increasing bandwidth of

[*] Supported by U.S. National Science Foundation under grant CCF-1116781.

the Internet make memory access a critical bottleneck for high-performance packet classification. State-of-the-art multi-core optimized microarchitectures [7], [8] deliver a number of new and innovative features that can improve memory access performance. The increasing gap between processor speed and memory speed was bridged by caches and instruction level parallelism (ILP) techniques [9]. For cache hits, latencies scale with reductions in cycle time. A cache hit typically introduces a latency of two or three clock cycles, even when the processor frequency increases. The cache misses are overlapped with other misses as well as useful computation using ILP. These features make multi-core processors an attractive platform for low-latency network applications. Efficient parallel algorithms are also needed on multi-core processors to improve the performance of network applications.

In this paper, we focus on improving the performance of packet classification with respect to throughput and latency on multi-core processors. Specifically, we use decomposition-based approaches on state-of-the-art multi-core processors and conduct a thorough comparison for various implementations. The rest of the paper is organized as follows. Section 2 formally describes the background and related work, and Section 3 covers the details of the four decomposition-based algorithms. Section 4 summarizes performance results, and Section 5 concludes the paper.

2 Background

2.1 Multi-field Packet Classification

Multi-field packet classification problem [10] requires five fields to be examined against the rules: 32-bit source IP address (SIP), 32-bit destination IP address (DIP), 16-bit source port number (SP), 16-bit destination port number (DP), and 8-bit transport layer protocol (PROT). In SIP and DIP fields, longest prefix match is performed on the packet header. In SP and DP fields, the matching condition of a rule is a range match. The PROT field only requires exact value to be matched. We denote this problem as the classic packet classification problem in this paper. A new version of packet classification is the OpenFlow packet classification [2] where a larger number of fields are to be examined. In this paper we focus on the classic packet classification, although our solution techniques can also be extended to the OpenFlow packet classification.

Each packet classification engine maintains a rule set. In this rule set, each rule has a rule ID, the matching criteria for each field, an associated action (ACT), and/or priority (PRI). For an incoming packet, a match is reported if all the five fields match a particular rule in the rule set. Once the matching rule is found for the incoming packet, the action associated with that rule is performed on the packet. We show an example rule set consisting of 8 rules in Table 1; the typical rule set size (denoted as N) ranges from 50 to 1K [10].

A packet can match multiple rules. If only the highest priority one needs to be reported, it is called *best-match* packet classification [11]; if all the matching rules need to be reported, it is a *multi-match* packet classification. Our implementation is able to output both results.

Table 1. Example rule set

ID	SA	DA	SP	DP	PROT	PRI	ACT
1	175.77.88.155/32	119.106.158.230/32	0 - 65535	6888 - 6888	0x06	1	Act 0
2	175.77.88.6/32	36.174.239.222/32	0 - 65535	1704 - 1704	0x06	2	Act 1
3	175.77.88.4/32	36.174.239.222/32	0 - 65535	1708 - 1708	0x06	3	Act 0
4	95.105.143.51/32	39.240.26.229/32	0 - 65535	1521 - 1521	0x06	4	Act 2
5	95.105.143.51/32	204.13.218.182/32	0 - 65535	0 - 65535	0x01	5	Act 3
6	152.175.65.32/28	248.116.141.0/28	24032 - 24032	123 - 123	0x11	5	Act 5
7	17.21.12.0/23	224.0.0.0/5	0 - 65535	0 - 65535	0x00	6	Act 4
8	233.117.49.48/28	233.117.49.32/28	750 - 750	123 - 123	0x11	7	Act 3

2.2 Related Work

Most of packet classification algorithms on general purpose processors fall into two categories: *decision-tree based* and *decomposition based* algorithms.

The most well-known decision-tree based algorithms are HiCuts [12] and HyperCuts [13] algorithms. The idea of decision-tree based algorithms is that each rule defines a sub-region in the multi-dimensional space and a packet header is viewed as a point in that space. The sub-region which the packet header belongs to is located by cutting the space into smaller sub-regions recursively. However, searching in a large tree is hard to parallelize and it requires too many memory accesses. Therefore it is still challenging to achieve high performance using efficient search tree structure on state-of-the-art multi-core processors.

Decomposition based algorithms usually contain two steps. The first step is to *search* each field individually against the rule set. The second step is to *merge* the partial results from all the fields. The key challenge of these algorithms is to parallelize individual search processes and handle merge process efficiently. For example, one of the decomposition based approaches is the Bit Vector (BV) approach [17]. The BV approach is a specific technique in which the lookup on each field returns an N-bit vector. Each bit in the bit vector corresponds to a rule. A bit is set to "1" only if the input matches the corresponding rule in this field. A bit-wise logical AND operation gathers the matches from all fields in parallel.

The BV-based approaches can achieve 100 Gbps throughput on FPGA [19], but the rule set size they support is typically small (less than 10K rules). Also, for port number fields (SP and DP), since BV-based approaches usually require rules to be represented in ternary strings, they suffers from range expansion when converting ranges into prefixes [20].

Some recent work has proposed to use multi-core processors for packet classification. For example, the implementation using HyperSplit [14], a decision-tree based algorithm, achieves a throughput of more than 6Gbps on the Octeon 3860 multi-core platform. However, the decomposition based approaches on state-of-the-art multi-core processors have not been well studied and evaluated.

3 Algorithms

We denote the rules in the classification rule set as *original rules*. Given a 5-field rule set, we present the procedure of our decomposition-based approaches in 3 phases:

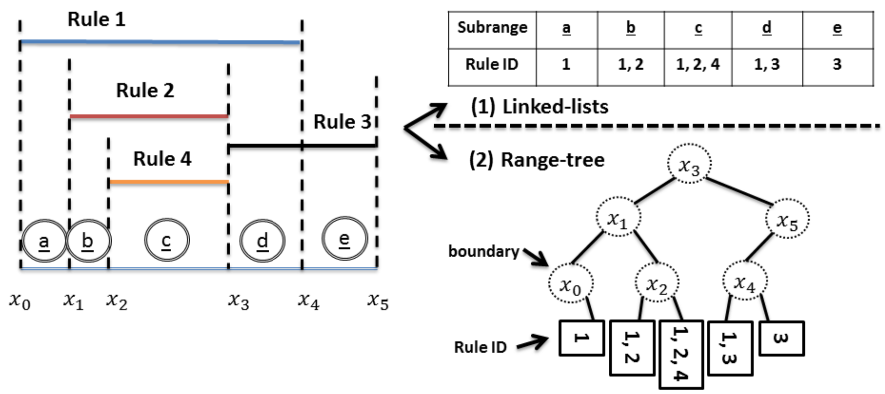
- Preprocess: The prefixes in IP address fields or exact values in PROT field are translated into ranges first; then all the rules are projected onto each field to produce a set of non-overlapping subranges in each field. We denote the rule specified by a subrange as a *unique rule*. Rules having the same value in a specific field are mapped into the same unique rule in this field. For example, the 8 rules in Table 1 can be projected into 5 unique rules in the SP field: [0, 749], [750, 750], [751, 24031], [24032, 24032], and [24033, 65535]. We either construct *linked-lists* using the unique rules for classification, or build a *range-tree* using the unique rules. We show an example for preprocessing 4 rules in a specific field in Figure 1.
- Search: The incoming packet header is also split into 5 fields, where each of the header field is searched individually and independently. We employ *linear search* or *range-tree search* for each individual field, and we record the partial matching results of each field by a *rule ID set* or a *Bit Vector* (BV).
- Merge: The partial results from different fields are merged into the final result. For partial results represented by a rule ID set, merge operation is performed using linear merge; for partial results represented by a BV, merge operation is performed using a bitwise logical AND.

Depending on the types of search phase and the representation of partial results, we have various implementations. Section 3.1 and 3.2 introduce the algorithms for linear search and range-tree search, respectively, while Section 3.3 and 3.4 cover the set representation and BV representation of partial results. Section 3.5 summarizes all the implementation.

3.1 Linear Search

A linear search can be performed in each field against unique rules. We denote the match of each unique rule as a *unique match*. We consider linear search due to the following reasons:

- Linear search can be used as a baseline to compare various decomposition-based implementations on multi-core processors.
- The number of unique rules in each field is usually less than the total number of the original rules [15].
- For multi-match packet classification problem (see Section 2), it requires *O(N)* memory to store a rule set consisting of *N* rules, while it also requires *O(N)* time to obtain all the matching results in the worst case. Linear search is still an attractive algorithm from this viewpoint.
- Linear search results in regular memory access patterns; this in turn can lead to good cache performance.



Subrange	a	b	c	d	e
Rule ID	1	1, 2	1, 2, 4	1, 3	3

Fig. 1. Preprocessing the rule set to (1) linked-lists or (2) a balanced binary range-tree

To optimize the search, we construct linked-lists for linear search. Linked-list is a data structure which specifies a unique rule and its associated original rules. In Figure 1, we show an example of constructing linked-lists from original rules in a specific field. As shown in Figure 1, all the 4 rules are preprocessed into 5 linked-lists due to the overlap between different original rules; during the search phase, the incoming packet is examined using the unique rules (comparing the input value sequentially with the subrange boundaries $x_0, x_1, \ldots, x_5$); the partial results are then stored for merge phase using either a rule ID set (Section 3.3), or a BV (Section 3.4).

3.2 Range-Tree Search

The key idea of range-tree search is to construct a balanced range tree [16] for efficient tree-search; the range-tree search can be applied to all the fields in parallel.

To construct a range-tree, we first translate prefixes or exact values in different packet header fields into ranges. Overlapping ranges are then flattened into a series of non-overlapping subranges as shown in Figure 1. A balanced binary search tree (range-tree) is constructed using the boundaries of those subranges.

For example, in Figure 1, the value in a specific field of the input packet header is compared with the root node (boundary x_3) of the range tree first. Without loss of generality, we use inclusive lower bound and exclusive upper bound for each subrange. Hence there can be only two outcomes from the comparison: (1) the input value is less than x_3, or (2) the input value is greater than or equal to x_3. In the first case, the input value is then compared with the left child (x_1) of the root node; in the second case, the input value is compared with the right child (x_5) of the root node. This algorithm is applied iteratively until the input value is located in a leaf node representing a subrange. Then the partial results can be extracted for future use in the merge phase.

This preprocess explicitly gives a group of non-overlapping subranges; the input can be located in a subrange through the binary search process. However, as shown in Figure 1, each subrange may still correspond to one or more valid matches.

3.3 Set Representation

After the search in each individual field is performed, we can represent each partial result using a set. As shown in Figure 2, for the input located in the subrange c, a set consisting of 3 rule IDs {1, 2, 4} is used to record the partial matching result in a field. We denote such a set as a *rule ID set*. We maintain the rule IDs in a rule ID set in sorted order. Thus, for the partial result represented using a rule ID set, all the rule IDs are arranged in ascending (or descending) order.

As shown in Algorithm 1, the merging of those partial results is realized by linear merge efficiently: we iteratively eliminate the smallest rule ID value of all 5 fields unless the smallest value appears in all the 5 fields. A common rule ID detected during this merge process indicates a match. The correctness of Algorithm 1 can be easily proved; this algorithm is similar as the merge operation for two sorted lists in merge sort, except 5 rule ID sets need to be merged.

We show an example of merging 5 rule ID sets in Figure 2. Initially we have 5 rule ID sets as the partial results from 5 fields. In the first iteration, since the rule IDs are stored in each rule ID set in ascending order, the first rule IDs in 5 rule ID sets (smallest rule IDs, namely, Rule 1 in SIP field, Rule 2 in DIP field, Rule 1 in SP field, Rule 3 in DP field, and Rule 3 in PROT field) are compared; since they are not the same, we delete the minimum ones from the rule ID sets: Rule 1 is deleted from the SIP rule ID set, and Rule 1 is deleted from the SP rule ID set. We iteratively apply this algorithm to delete minimum rule IDs in each iteration, unless all the minimum rule IDs of 5 fields are the same; a common rule ID appearing in all 5 fields (such as Rule 3 in Figure 2) indicates a match between the packet header and the corresponding rule. We record the common rule IDs as the final result.

```
Algorithm 1: (Set $S_0$) MERGE (Set $S_1$, Set $S_2$, ..., Set $S_5$)
if any of the input sets is null then
return (null)
else
 for non-null input Set $S_i$ do in parallel
   $a_i$ → first element of $S_i$
   end for
   while ($a_i$ is not the last element of $S_i$) do
if not all $a_i$'s are equal then
  $S_i$ → delete the minimum $a_i$ from $S_i$
  MERGE (Set $S_1$, Set $S_2$, ..., Set $S_5$)
else
  push $a_i$ into set $S_0$
  $S_i$ → delete $a_i$ from $S_i$ for all $i$
  MERGE (Set $S_1$, Set $S_2$, ..., Set $S_5$)
      end if
   end while
   return (Set $S_0$)
  end if
```

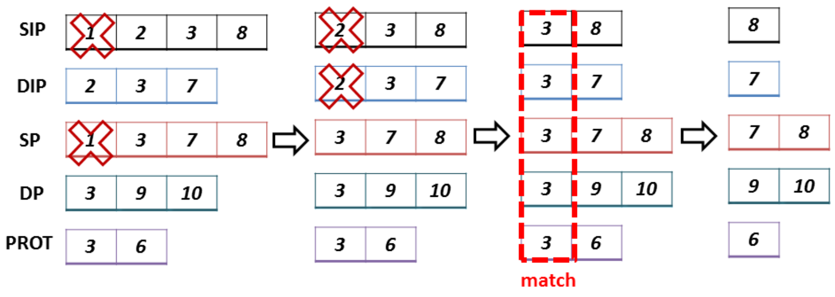


Fig. 2. Merging 5 rule ID sets

3.4 BV Representation

Another representation for the partial results is the BV representation [17]. For each unique rule/subrange, a BV is maintained to store the partial results. For a rule set consisting of N rules, the length of the BV is N for each unique rule/subrange, with each bit corresponding to the matched rules set to 1. For example, in Figure 1, the matched rule IDs for the subrange c can be represented by a BV "1101", indicating a match for this subrange corresponds to Rule 1, Rule 2 and Rule 4.

Both linear search and range-tree search generate 5 BVs as partial results for 5 fields, respectively. The partial results can be merged by bitwise AND operation to produce the final result. In the final result, every bit with value 1 indicates a match in all the five fields; we report either all of their corresponding rules (multi-match) or only the rule with the highest priority (best-match).

3.5 Implementations

In summary, we have four implementations: (1) Linear search in each field, with partial results represented as rule ID Sets (LS), (2) Range-tree search with partial results recorded in rule ID Sets (RS), (3) Linear search with BV representation (LBV) and (4) Range-tree search with BV representation (RBV).

We denote the total number of original rules as N, the number of unique rules as N_1 and the maximum number of original rules associated with a unique rule as N_2. Table 2 summarizes the computation time complexity for the four approaches.

Table 2. Summary of various approaches

Approach	Search	Merge
LS	$O(N_1)$	$O(N_2 \log(N))$
RS	$O(\log(N_1))$	$O(N_2 \log(N))$
LBV	$O(N_1)$	$O(N_2)$
RBV	$O(\log(N_1))$	$O(N_2)$

4 Performance Evaluation and Summary of Results

4.1 Experimental Setup

Since our approaches are not platform-dependent, we conducted the experiments on a 2× AMD Opteron 6278 processor and a 2× Intel Xeon E5-2470 processor. The AMD processor has 16 cores, each running at 2.4GHz. Each core is integrated with a 16KB L1 data cache, 64KB L1 instruction cache and a 2MB L2 cache. A 60MB L3 cache is shared among all 16 cores. The processor has access to 128GB DDR3 main memory through an integrated memory controller running at 2GHz. The Intel processor has 16 cores, each running at 2.3GHz. Each core has a 32KB L1 data cache, 32KB L1 instruction cache and a 256KB L2 cache. All 16 cores share a 20MB L3 cache.

We implemented all the 4 approaches using Pthreads on openSUSE 12.2. We analyzed the impact of data movement in our approaches. We also used *perf*, a performance analysis tool in Linux, to monitor the hardware and software events such as the number of executed instructions, the number of cache misses and the number of context switches.

We generated synthetic rule sets using the same methodology as in [18]. We varied the rule set size from 1K to 64K to study the scalability of our approaches. We used processing *latency* and overall *throughput* as the main performance metrics. We define overall throughput as the aggregated throughput in Gbps of all parallel packet classifiers. We define the processing latency as the average processing time elapsed during packet classification for a single packet. We also examined the relation between the number of threads per core and context switch frequency to study their impact on the overall performance.

4.2 Data Movement

To optimize our implementation, we first conduct two groups of RBV experiments with 1K rule set: (Design 1) 5 cores process 5 packets in parallel, each processing a single packet independently; (Design 2) all the 5 cores process a single packet, each processing one packet header field. In Design 1, partial results from the five fields of a packet stay in the same core and the search phase requires 5 range trees to be used

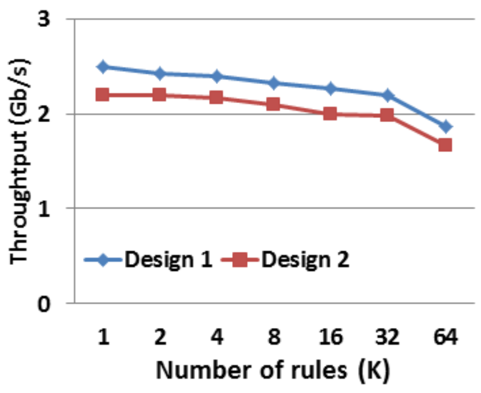
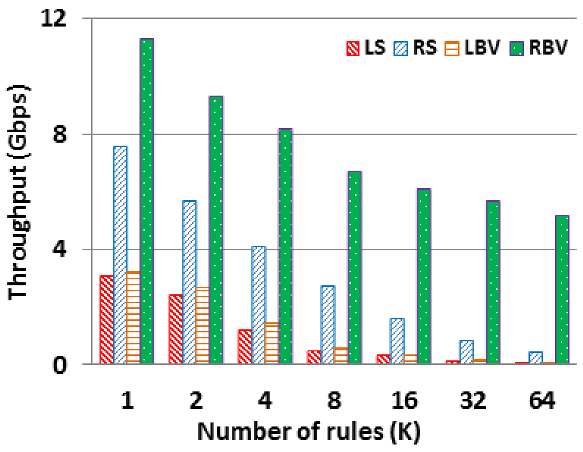


Fig. 3. Data movement (Design 1: 5 cores, 5 packets; Design 2: 5 cores, 1 packet)

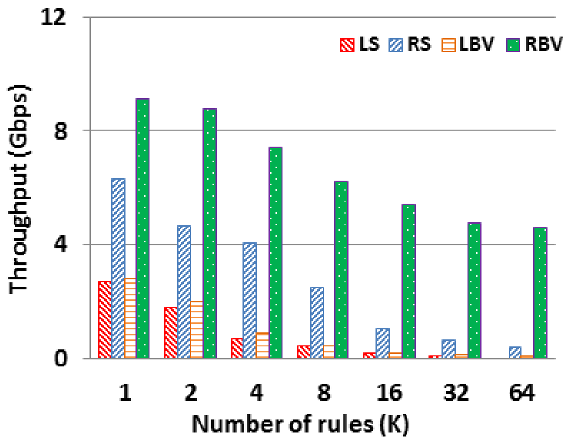
in the same core. In Design 2, all the threads in a single core perform search operation in the same packet header field; the merge phase requires access to partial results from different cores. Our results in Figure 3 show that Design 1 has in average 10% more throughput than Design 2. In Design 2, a large amount of data move between cores, making it less efficient than Design 1. We also have similar observations in LS, RS, and LBV approaches. Therefore, we use Design 1 (a packet only stays in one core) for all of the 4 approaches, where partial results are accessed locally.

4.3 Throughput and Latency

Figure 4 and Figure 5 show the throughput and latency performance with respect to various sizes of the rule set for all the four approaches. For each approach, the

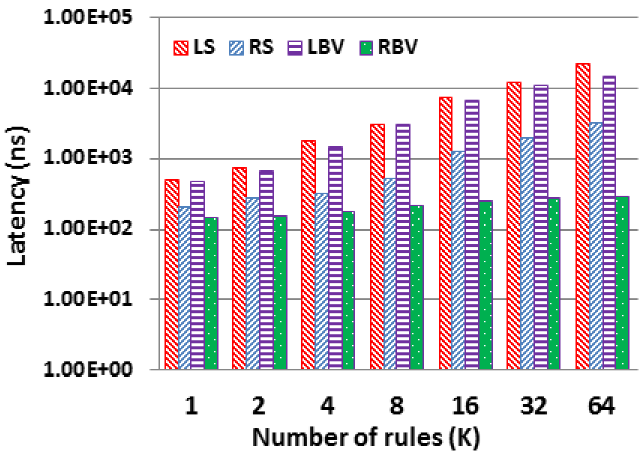


(a) Throughput on the AMD processor

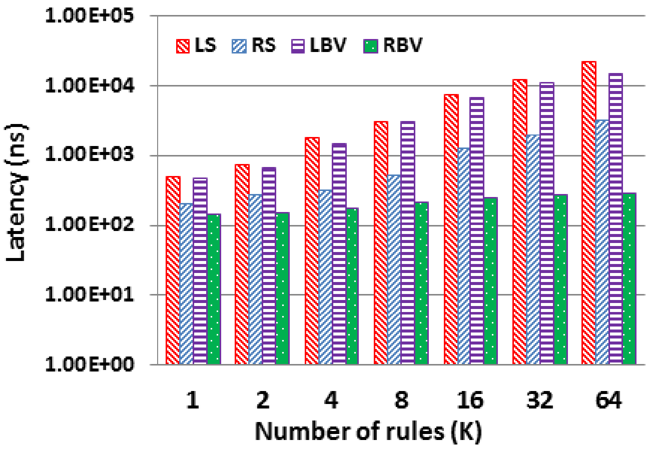


(b) Throughput on the Intel processor

Fig. 4. Throughput with respect to the number of rules (*N*)



(a) Latency on the AMD processor



(b) Latency on the Intel processor

Fig. 5. Latency with respect to the number of rules (*N*)

experiments are conducted on both the AMD and the Intel platforms. We have the following observations:

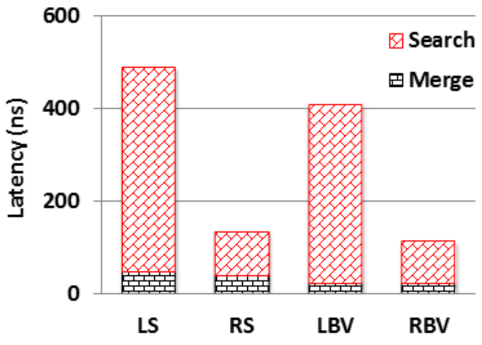
- The performance degrades as the number of rules increases. This is because when the rule set becomes larger, the number of unique rules also increases; this leads to an increase of both the search time and the merge time.
- Range-tree based approaches suffer less performance degradation when the number of rules increases. Note the time complexity for range-tree search is $O(\log N_1)$, while the time complexity for linear search is $O(N_1)$. For a balanced binary range-tree, when the rule set size doubles, one more tree level has to be searched, while linear search requires double the amount of search time. We show the breakdown of the overall execution time in terms of search and merge times in Section 4.4.

- Using set representation in merge phase (LS and RS) is not as efficient as using BV representation (LBV and RBV). Merging multiple rule ID sets requires O(N_2 log(N)) time, while it takes O(N_2) time to merge bit vectors using bitwise AND operations.

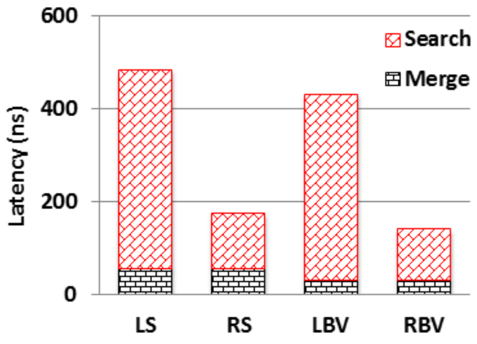
4.4 Search Latency and Merge Latency

We show the latency of the search and merge phases in Figure 6. We have the following observations:

- Search time contributes more to the total classification latency than the merge time, since search operations are more complex compared with the merge operations.
- As shown in Figure 5 and Figure 6, we achieve similar performance on the AMD and the Intel multi-core platforms. The performance of all the implementations on Intel machine is slightly worse than the performance on the AMD machine. Note that the Intel machine has a lower clock rate.



(a) Latency on the AMD processor



(b) Latency on the Intel processor

Fig. 6. Breakdown of the latency per packet on (a) the AMD multi-core processor and (b) the Intel multi-core processors (1K rule set, 5 threads per core)

4.5 Cache Performance

To explore further why the performance deteriorates as the rule size grows, we measure the cache performance. We show the number of cache misses per 1K packets under various scenarios on the AMD processor in Figure 7. As can be seen:

- Linear search based approaches result in more cache misses than the range-tree search based approaches because the linear search requires all the unique rules to be compared against the packet header.
- Approaches based on set representation have more cache misses than the BV based approaches. Set representation requires log(*N*) bits for each rule ID, while the BV representation only requires 1 bit to indicate the match result for each rule. Hence BV representation of the matched rules can be fit in the cache of a multi-core processor more easily.
- The overall performance is consistent with the cache hits on the multi-core processors. RBV introduces the least amount of cache misses and achieves the highest overall throughput with minimum processing latency.

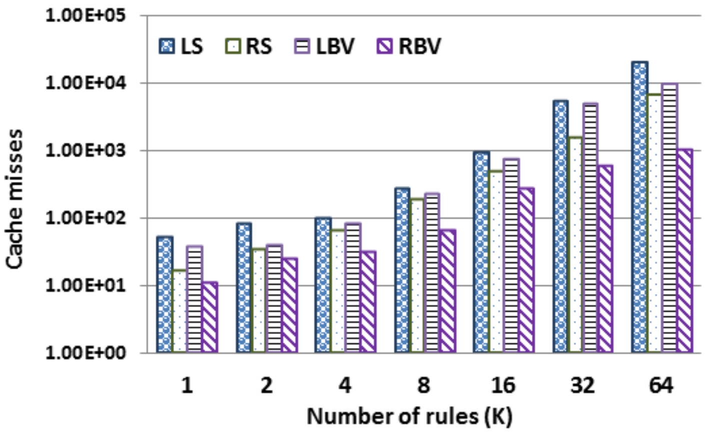
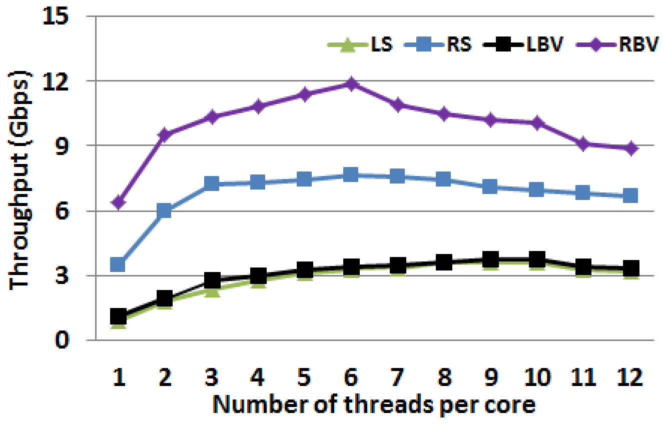


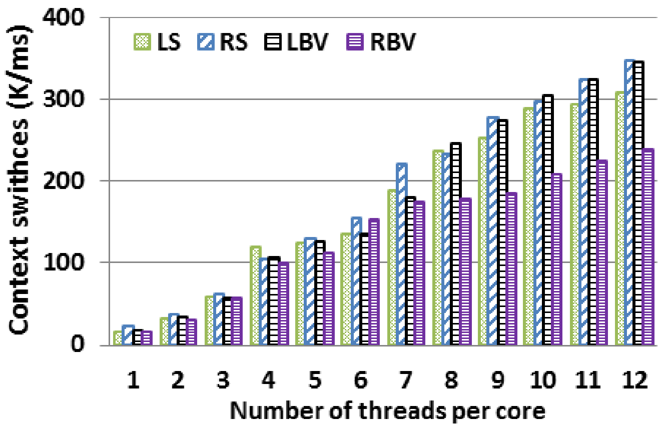
Fig. 7. Number of L2 cache misses per 1K packets on the AMD processor

4.6 Number of Threads per Core (*T*)

The number of threads per core also has an impact on the performance for all four approaches. Our results, as shown in Figure 8a, indicate the performance of RS and RBV goes up as the number of threads per core (*T*) increases from 1 to 6. Once *T* exceeds 6 the throughput begins to degrade. For LS and LBV, the performance keeps increasing until *T* reaches 10. Reasons for performance degradation include saturated resource consumption of each core and the extra amount of overhead brought by the context switch mechanism. Figure 8b illustrates the relation between context switch frequency and the number of threads per core. When *T* increases, context switches happen more frequently, and switching from one thread to another requires a large amount of time to save and restore states.



(a) Throughput vs. number of threads per core



(b) Context switch vs. number of threads per core

Fig. 8. Performance vs. number of threads per core

We also evaluate $T > 13$ in our experiments, and we observe the performance drops dramatically for all four approaches. The performance of the approaches based on linear search deteriorates for $T > 10$, while the performance of range-tree based approaches deteriorates for $T > 6$. We explain the underlying reasons why the performance of range-tree based approaches degrades at a smaller T as follows:

- The different threads in the same core may go through different paths from the root to the leaf nodes, when range-tree search is performed. For a large number of threads in the same core, cache data cannot be efficient shared among different threads; this leads to performance degradation at a smaller value of T.
- For linear search, since all the threads use the same data and have regular memory access patterns in the search phase, the common cache data can be

efficiently shared among different threads. However, as can be seen from Figure 8b, the frequent context switches still adversely affect the overall throughput performance. This in turn results in performance degradation at a larger value of *T*.

5 Conclusion and Future Work

We implemented and compared four decomposition-based packet classification algorithms on state-of-the-art multi-core processors. We explored the impact of the rule set size (*N*), the number of threads per core (*T*), and the communication between cores on performance with respect to latency and throughput.

We also examined the cache performance and conducted experiments on different platforms. Our experimental results show that range search is much faster than linear search and is less influenced by the size of rule set. We also find that BV representation based approaches are more efficient than set representation based approaches.

In the future, we plan to apply our approaches on OpenFlow packet classification where more packet fields are required to be examined. We will also implement decision-tree based and hashed based packet classification algorithms on multi-core platforms.

References

1. Naous, J., Erickson, D., Covington, G.A., Appenzeller, G., McKeown, N.: Implementing an OpenFlow Switch on the NetFPGA Platform. In: Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2008, pp. 1–9 (2008)
2. Brebner, G.: Softly Defined Networking. In: Proc. of the 8th ACM/IEEE Symp. on Architectures for Networking and Communications Systems, ANCS 2012, pp. 1–2 (2012)
3. Yu, F., Katz, R.H., Lakshman, T.V.: Efficient Multimatch Packet Classification and Lookup with TCAM. IEEE Micro 25(1), 50–59 (2005)
4. Jiang, W., Wang, Q., Prasanna, V.K.: Beyond TCAMs: an SRAM based parallel multi-pipeline architecture for terabit IP lookup. In: Proc. IEEE INFOCOM, pp. 1786–1794 (2008)
5. Jedhe, G.S., Ramamoorthy, A., Varghese, K.: A Scalable High Throughput Firewall in FPGA. In: Proc. of IEEE Symposium on Field Programmable Custom Computing Machines, FCCM, pp. 802–807 (2008)
6. Koponen, T.: Software is the Future of Networking. In: Proc. of the 8th ACM/IEEE Symp. on Architectures for Networking and Communications Systems, ANCS, pp. 135–136 (2012)
7. AMD Multi-Core Processors, `http://www.computerpoweruser.com/articles/archive/c0604/29c04/29c04.pdf`
8. Intel Multi-Core Processors: Making the Move to Quad-Core and Beyond, `http://www.cse.ohio-state.edu/~panda/775/slides/intel_quad_core_06.pdf`
9. Multicore Computing- the state of the art, `http://eprints.sics.se/3546/1/SMI-MulticoreReport-2008.pdf`

10. Gupta, P., McKeown, N.: Packet classification on multiple fields. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM, pp. 147–160 (1999)
11. Jiang, W., Prasanna, V.K.: A FPGA-based Parallel Architecture for Scalable High-Speed Packet Classification. In: 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP, pp. 24–31 (2009)
12. Gupta, P., McKeown, N.: Packet Classification using Hierarchical Intelligent Cuttings. In: IEEE Symposium on High Performance Interconnects, HotI (1999)
13. Singh, S., Baboescu, F., Varghese, G., Wang, J.: Packet Classification using Multidimensional Cutting. In: ACM SIGCOMM, pp. 213–224 (2003)
14. Liu, D., Hua, B., Hu, X., Tang, X.: High-performance Packet Classification Algorithm for Any-core and Multithreaded Network Processor. In: Proc. CASES (2006)
15. Taylor, D.E., Turner, J.S.: Scalable Packet Classification using Distributed Crossproducing of Field Labels. In: Proc. IEEE INFOCOM, pp. 269–280 (2005)
16. Zhong, P.: An IPv6 Address Lookup Algorithm based on Recursive Balanced Multi-way Range Trees with Efficient Search and Update. In: Proc. of International Conference on Computer Science and Service System, CSSS 2011, pp. 2059–2063 (2011)
17. Lakshman, T.V.: High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In: ACM SIGCOMM, pp. 203–214 (1998)
18. Pong, F., Tzeng, N.-F., Tzeng, N.-F.: HaRP: Rapid Packet Classification via Hashing Round-Down Prefixes. IEEE Transactions on Parallel and Distributed Systems 22(7), 1105–1119 (2011)
19. Jiang, W., Prasanna, V.K.: Field-split Parallel Architecture for High Performance Muti-match Packet Classification using FPGAs. In: Proc. of the 21st Annual Symp. on Parallelism in Algorithms and Arch., SPAA, pp. 188–196 (2009)
20. Srinivasan, V., Varghese, G., Suri, S., Waldvogel, M.: Fast and Scalable Layer Four Switching. In: Proc. ACM SIGCOMM, pp. 191–202 (1998)

Slot Selection Algorithms in Distributed Computing with Non-dedicated and Heterogeneous Resources

Victor Toporkov[1], Anna Toporkova[2], Alexey Tselishchev[3], and Dmitry Yemelyanov[1]

[1] National Research University "MPEI",
ul. Krasnokazarmennaya, 14, Moscow, 111250, Russia
{ToporkovVV,YemelyanovDM}@mpei.ru
[2] National Research University Higher School of Economics,
Moscow State Institute of Electronics and Mathematics,
Bolshoy Trekhsvyatitelsky per., 1-3/12, Moscow, 109028, Russia
atoporkova@hse.ru
[3] European Organization for Nuclear Research (CERN),
Geneva, 23, 1211, Switzerland
Alexey.Tselishchev@cern.ch

Abstract. In this work, we introduce slot selection and co-allocation algorithms for parallel jobs in distributed computing with non-dedicated and heterogeneous resources (clusters, CPU nodes equipped with multicore processors, networks etc.). A single slot is a time span that can be assigned to a task, which is a part of a job. The job launch requires a co-allocation of a specified number of slots starting synchronously. The challenge is that slots associated with different resources of distributed computational environments may have arbitrary start and finish points that do not match. Some existing algorithms assign a job to the first set of slots matching the resource request without any optimization (the first fit type), while other algorithms are based on an exhaustive search. In this paper, algorithms for effective slot selection of linear complexity on an available slots number are studied and compared with known approaches. The novelty of the proposed approach consists of allocating alternative sets of slots. It provides possibilities to optimize job scheduling.

Keywords: Distributed computing, economic scheduling, resource management, slot, job, task, batch.

1 Introduction

Economic mechanisms are used to solve problems like resource management and scheduling of jobs in a transparent and efficient way in distributed environments such as cloud computing and utility Grid [1, 2]. A resource broker model is decentralized, well-scalable and application-specific [2-4]. It has two parties: node owners and brokers representing users. The simultaneous satisfaction of various application optimization criteria submitted by independent users is not possible due to several reasons [2] and also can deteriorate such quality of service rates as total execution time of a sequence of jobs or overall resource utilization. Another model is related to virtual organizations (VO) [5-7] and metascheduling with central schedulers or a Meta-Broker [2] providing job-flow level scheduling and optimization. VOs naturally restrict the scalability, but uniform rules for allocation and consumption of resources make it possible to improve the efficiency of resource usage and to find a tradeoff between contradictory interests of different participants. In [6], we have proposed a hierarchical model of resource management system which is functioning within a VO. Resource management is implemented using a structure consisting of a metascheduler and subordinate job schedulers that interact with batch job processing systems. The significant difference between the approach proposed in [6] and well-known scheduling solutions for distributed environments such as Grids [1-5, 8, 9], e.g., gLite Workload Management System [8], where Condor is used as a scheduling module, is the fact that the scheduling strategy is formed on a basis of efficiency criteria. They allow reflecting economic principles of resource allocation by using relevant cost functions and solving a load balancing problem for heterogeneous resources. At the same time, the inner structure of the job is taken into account when the resulting schedule is formed. The metascheduler [5-7] implements the economic policy of a VO based on local resource schedules. The schedules are defined as sets of slots coming from resource managers or schedulers in the resource domains. During each scheduling cycle the sets of available slots are updated based on the information from local resource managers. Thus, during every cycle of the job batch scheduling [6] two problems have to be solved: 1) selecting an alternative set of slots (alternatives) that meet the requirements (resource, time, and cost); 2) choosing a slot combination that would be the efficient or optimal in terms of the whole job batch execution in the current cycle of scheduling. To implement this scheduling scheme, first of all, one needs to propose the algorithm for finding sets of alternative executions. An optimization technique for the second phase of this scheduling scheme was proposed in [6, 7].

The scheduling problem in Grid is NP-hard due to its combinatorial nature and many heuristic-based solutions have been proposed. In [4] heuristic algorithms for slot selection, based on user-defined utility functions, are introduced. NWIRE system [4] performs a slot window allocation based on the user defined efficiency criterion under the maximum total execution cost constraint. However, the optimization occurs only on the stage of the best found offer selection. First fit slot selection algorithms (backtrack [10] and NorduGrid [11] approaches) assign any job to the first set of slots matching the resource request conditions, while other algorithms use an exhaustive search [2, 12, 13] and some of them are based on a linear integer programming (IP) [2, 12] or mixed-integer programming (MIP) model [13]. Moab scheduler [14] implements the backfilling algorithm and during a slot window search does not take into account any additive constraints such as the minimum required storage volume or the maximum allowed total allocation cost. Moreover, backfilling does not support environments with non-dedicated resources and its execution time grows substantially with the increase of the slot numbers. Assuming that every CPU node has at least one local job scheduled, the backfilling algorithm has quadratic complexity in terms of the slot number. In our previous works [15-17], two algorithms for slot selection AMP and ALP that feature linear complexity $O(m)$, where m is the number of available time-slots, were proposed. Both algorithms perform the search of the first fitting window without any optimization. AMP (**A**lgorithm based on **M**aximal job **P**rice), performing slot selection based on the maximum slot window cost, proved the advantage over ALP (**A**lgorithm based on **L**ocal **P**rice of slots) when applied to the above mentioned scheduling scheme. However, in order to accommodate an end user's job execution requirements, there is a need for a more precise slot selection algorithm to exploit during the first stage of the proposed scheduling scheme and to consider various user demands along with the VO resource management policy.

In this paper, we propose algorithms for effective slot selection based on user defined criteria that feature linear complexity on the number of the available slots during the job batch scheduling cycle. The novelty of the proposed approaches consists of allocating a number of alternative sets of slots (alternatives). The proposed algorithms can be used for both homogeneous and heterogeneous resources. The paper is organized as follows. Section 2 introduces a general scheme for searching alternative slot sets that are effective by the specified criteria. Then four implementations are proposed and considered. Section 3 contains simulation results for comparison of proposed and known algorithms. Section 4 summarizes the paper and describes further research topics.

2 General Scheme and Slot Selection Algorithms

In this section we consider a general scheme of an **A**lgorithm searching for **E**xtreme **P**erformance (AEP) and its implementation examples.

2.1 AEP Scheme

The launch of any job requires a co-allocation of a specified number of slots, as well as in the classic backfilling variation [14]. The target is to scan a list of m available slots and to select a window W of n parallel slots with a length of the required resource reservation time. The job resource requirements are arranged into a resource request containing a resource reservation time, characteristics of computational nodes (clock speed, RAM volume, disk space, operating system etc.) and the limitation on the selected window maximum cost. The total window cost is calculated as a sum of an individual usage cost of the selected slots. According to the resource request, it is required to find a window with the following description: n concurrent time-slots providing the resource performance rate and the maximal resource price per time unit F should be reserved for a time span t_s. The length of each slot in the window is determined by the performance rate of the node on which it is allocated. Thus, in the case of heterogeneous resources, as a result one has a window with a "rough right edge" (Fig. 1). The window search is performed on the list of all available slots sorted by their start time in ascending order (this condition is necessary to examine every slot in the list and for operation of search algorithms of linear complexity [15-17]). In addition, one can define a criterion crW on which the best matching window alternative is chosen: This can be a criterion for a minimum cost, a minimum execution runtime or, for example, a minimum energy consumption. The algorithm parses a ranged list of all available slots subsequently for all the batch jobs. Higher priority jobs are processed first [6].

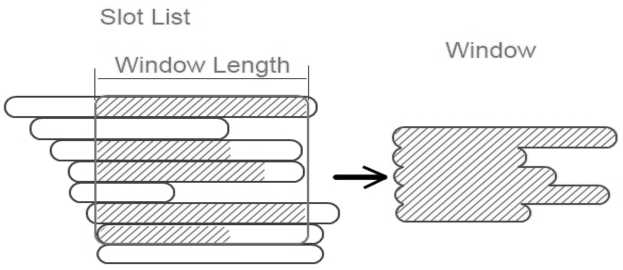


Fig. 1. Window with a "rough right edge"

Consider as an example the problem of selecting a window of size n with a total cost no more than S from the list of $m > n$ slots (in the case, when $m = n$ the selection is trivial). The maximal job budget is counted as $S = Ft_s n$. The current extended window consists of m slots $s_1, s_2, ..., s_m$. The cost of using each of the slots according to their required time length is: $c_1, c_2, ..., c_m$. Each slot has a numeric characteristic z_i in accordance to crW. The total value of these characteristics should be minimized in the resulting window.

Then the problem could be formulated as follows:

$$a_1 z_1 + a_2 z_2 + ... + a_m z_m \to \min,\ a_1 c_1 + a_2 c_2 + ... + a_m c_m \le S,$$

$$a_1 + a_2 + ... + a_m = n,\ a_r \in \{0, 1\},\ r = 1, ..., m.$$

Additional restrictions can be added, for example, considering the specified value of deadline. Finding the coefficients $a_1, a_2, ..., a_m$ each of which takes integer values 0 or 1 (and the total number of "1" values is equal to n), determine the window with the specified criteria crW extreme value. By combining the optimization criteria, VO administrators and users can form alternatives search strategies for every job in the batch [6, 7]. Users may be interested in their jobs total execution cost minimizing or, for example, in the earliest possible jobs finish time, and are able to affect the set of alternatives found by specifying the job distribution criteria. VO administrators in their turn are interested in finding extreme alternatives characteristics values (e.g., total cost, total execution time) to form more flexible and, possibly, more effective combination of alternatives representing a batch execution schedule. The time length of an allocated window W is defined by the execution time of the task that is using the slowest CPU node. The algorithm proposed is processing a list of all slots available during the scheduling interval ordered by a non-decreasing start time (see Fig. 1). This condition is required for a single sequential slot list scan and algorithm linear complexity on the number m of slots.

The AEP scheme for an effective window search by the specified criteria can be represented as follows:

```
/* Job - Batch job for which the search is performed ;
** windowSlots – a set (list) of slots representing the
window;*/
slotList = orderSystemSlotsByStartTime();
for(i=0; i< slotList.size; i++){
   nextSlot = slotList[i];
 if(!properHardwareAndSoftware(nextSlot))
  continue; // The slot does not meet the requirements
 windowSlotList.add(nextSlot);
 windowStart = nextSlot.startTime;
 for(j=0; j<windowSlots.size; j++){
  wSlot = windowSlots[j];
  minLength = wSlot.Resource.getTime(Job);
  if((wSlot.EndTime – windowStart) < minLength)
    windowSlots.remove(wSlot);
 }
 if(windowSlots.size >= Job.m){
   curWindow = getBestWindow(windowSlots);
   crW = getCriterion(curWindow);
   if(crW > maxCriterion){
    maxCriterion = crW;
     bestWindow = curWindow;
   }
 }
}
```

Finally, a variable `bestWindow` will contain an effective window by the given criterion `crW`.

2.2 AEP Implementation Examples

The need to choose alternative sets of slots for every batch job increases the complexity of the whole scheduling scheme [6]. With a large number of the available slots the algorithm execution time may become inadequate. Though it is possible to mention some typical optimization problems, based on the AEP scheme that can be solved with a relatively decreased complexity. These include problems of total job cost, runtime minimizing, the window formation with the minimal start/finish time.

Consider the procedure for *minimizing a window start time*. The difference with the general AEP scheme is that the first suitable window will have the earliest possible start time. Indeed, if at some step i of the algorithm (after the i-th slot is added) the suitable window can be formed, then the windows, formed at the further steps will be guaranteed to have the start time that is not earlier (according to the ordered list of available slots, only slots with non-decreasing start time will be taken into consideration). This procedure can be reduced to finding a set of the first n parallel slots the total cost of which does not exceed the budget limit S. This description coincides the AMP scheme considered in previous works [15-17]. Thus AEP is naturally an extension of AMP, and AMP is the particular case of the whole AEP scheme performing only the start time optimization. Further we will use AMP abbreviation as a reference to the window start time minimization procedure. It is easy to provide the implementation of the algorithm of finding a window with *the minimum total execution cost.* For this purpose in the AEP search scheme n slots with the minimum sum cost should be chosen. If at each step of the algorithm a window with the minimum sum cost is selected, at the end the window with the best value of the criterion crW will be guaranteed to have overall minimum total allocation cost at the given scheduling interval.

The problem to find a window with *the minimum runtime* is more complicated. Given the nature of determining a window runtime, which is equal to the length of the longest slot (allocated on the node with the least performance level), the following algorithm may be proposed:

```
orderSlotsByCost(windowSlots);
resultWindow = getSubList(0,n, windowSlots);
extendWindow = getSubList(n+1,m, windowSlots);
while(extendWindow.size > 0){
  longSlot = getLongestSlot(resultWindow);
  shortSlot = getCheapestSlot(extendWindow);
  extendWindow.remove(shortSlot);
  if((shortSlot.size < longSlot.size)&&
   (resultWindow.cost + shortSlot.cost < S)){
      resultWindow.remove(longSlot);
      resultWindow.add(shortSlot);
  }
}
```

As a result, the suitable window of the minimum time length will be formed in a variable `resultWindow.` The algorithm described consists of the consecutive attempts to substitute the longest slot in the forming window (the `resultWindow` variable) with another shorter one that will not be too expensive. In case when it is impossible to substitute the slots without violating the constraint on the maximum window allocation cost, the current `resultWindow` configuration is declared to have the minimum runtime. Implementing this algorithm of window selection at each step of the AEP scheme allows finding a suitable window with the minimum possible runtime at the given scheduling interval. An algorithm for finding a window with *the earliest finish time* has a similar structure and can be described using the runtime minimizing procedure presented above. Indeed, the expanded window has a start time `tStart` equal to the start time of the last added suitable slot. The minimum finish time for a window on this set of slots is `(tStart + minRuntime)`, where `minRuntime` is the minimum window length. The value of `minRuntime` can be calculated similar to the runtime minimizing procedure described above. Thus, by selecting a window with the earliest completion time at each step of the algorithm, the required window will be allocated at the end of the slot list.

It is worth mentioning that all proposed AEP implementations have a linear complexity $O(m)$: algorithms "move" through the list of the m available slots in the direction of non-decreasing start time without turning back or reviewing previous steps.

3 Experimental Studies of Slot Selection Algorithms

The goal of the experiment is to examine AEP implementations: to analyze alternatives search results with different efficiency criteria, to compare the results with AMP and to estimate the possibility of using in real systems considering the algorithm executions time.

3.1 Algorithms and Simulation Environment

For the proposed AEP efficiency analysis the following implementations were added to the simulation model [6, 7]: 1) *AMP* – the algorithm for searching alternatives with the earliest start time. This scheme was introduced in works [15-17] and briefly described in section 2.2; 2) *MinFinish* – the algorithm for searching alternatives with the earliest finish time. It likewise involves finding a single alternative with the earliest finish time for each batch job (the procedure is described in section 2.2); 3) *MinCost* – the algorithm for searching a single alternative with the minimum total allocation cost on the scheduling interval; 4) *MinRunTime* – this algorithm performs a search for a single alternative with the minimum execution runtime (the window's runtime is defined as a length of the longest composing slot); 5) *MinProcTime* – this algorithm performs a search for a single alternative with the minimum total node execution time (defined as a sum of the composing slots' time lengths). It is worth mentioning that this implementation is simplified and does not guarantee an optimal result and only partially matches the AEP scheme, because a random window is selected; 6) *Common Stats, AMP* (further referred to as *CSA*) – the scheme for searching multiple alternatives using *AMP*. Similar to the general searching scheme [15-17], a set of suitable alternatives, disjointed by the slots, is allocated for each job. To compare the search results with the algorithms 1-5, presented above, only alternatives with the extreme value of the given criterion will be selected, so the optimization will take place at the selection process. The criteria include the minimum start time, the minimum finish time, the minimum total execution cost, the minimum runtime and the minimum processor time used.

Since the purpose of the considered algorithms is to allocate suitable alternatives, it makes sense to make the simulation apart from the whole general scheduling scheme, described in [6]. In this case, the search will be performed for a single predefined job. Thus during every single experiment a generation of a new distributed computing environment will take place while the algorithms described will perform the alternatives search for a single base job with the resource request defined in advance. A simulation framework [6, 7] was configured in a special way in order to study and to analyze the algorithms presented. The core of the system includes several components that allow generating the initial state of the distributed environment on the given scheduling interval, a batch with user jobs and implements the developed alternative search algorithms.

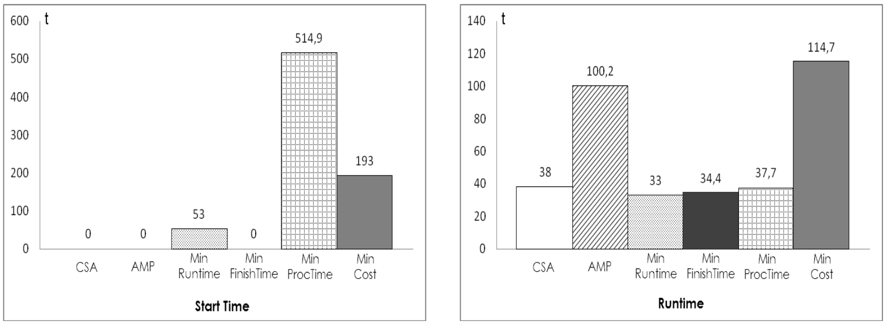
In each experiment a generation of the distributed environment that consists of 100 CPU nodes was performed. The relatively high number of the generated nodes has been chosen to allow *CSA* to find more slot alternatives. Therefore more effective alternatives could be selected for the searching results comparison based on the given criteria. The performance rate for each node was generated as a random integer variable in the interval [2; 10] with a uniform distribution. The resource usage cost was formed proportionally to their performance with an element of normally distributed deviation in order to simulate a free market pricing model [1-4]. The level of the resource initial load with the local and high priority jobs at the scheduling interval [0; 600] was generated by the hyper-geometric distribution in the range from 10% to 50% for each CPU node. Based on the generated environment the algorithms performed the search for a single initial job that required an allocation of 5 parallel slots for 150 units of time. The maximum total execution cost according to user requirements was set to 1500. This value generally will not allow using the most expensive (and usually the most efficient) CPU nodes.

3.2 Experimental Results

The results of the 5000 simulated scheduling cycles are presented in Fig. 2-6.

An average number of the alternatives found with *CSA* for a single job during the one scheduling cycle was 57. This value can be explained as the balance between the initial resource availability level and the user job resource requirements. Thus, the selection of the most effective windows by the given criteria was carried out among 57 suitable alternatives on the average. With the help of other algorithms only the single effective by the criterion alternative was found. Consider the average *start time* for the alternatives found (and selected) by the aforementioned algorithms (Fig. 2 (a)). *AMP*, *MinFinish* and *CSA* were able to provide the earliest job start time at the beginning of the scheduling interval (t = 0). The result was expected for *AMP* and *CSA* (which is essentially based on the multiple runs of the *AMP* procedure) since 100 available resource nodes provide a high probability that there will be at least 5 parallel slots starting at the beginning of the interval and can form a suitable window. The fact that the *MinFinish* algorithm was able to provide the same start time can be explained by the local tasks minimum length value, that is equal to 10. Indeed, the window start time at the moment t = 10 cannot provide the earliest finish time even with use of the most productive resources (for example the same resources allocated for the window with the minimal runtime). Average starting times of the alternatives found by *MinRunTime*, *MinProcTime* and *MinCost* are 53, 514.9 and 193 respectively.

The *average runtime* of the alternatives found (selected) is presented in Fig. 2 (b). The minimum execution runtime 33 was obviously provided by the *MinRunTime* algorithm. Though, schemes *MinFinish*, *MinProcTime* and *CSA* provide quite comparable values: 34.4, 37.7 and 38 time units respectively that only 4.2%, 14.2% and 15.1% longer. High result for the *MinFinish* algorithm can be explained by the "need" to complete the job as soon as possible with the minimum (and usually initial) start time. Algorithms *MinFinish* and *MinRunTime* are based on the same criterion selection procedure described in the section 2.2. However due to non-guaranteed availability of the most productive resources at the beginning of the scheduling interval, *MinRunTime* has the advantage. Relatively long runtime was provided by *AMP* and *MinCost* algorithms. For *AMP* this is explained by the selection of the first fitting (and not always effective by the given criterion) alternative, while *MinCost* tries to use relatively cheap and (usually) less productive CPU nodes.

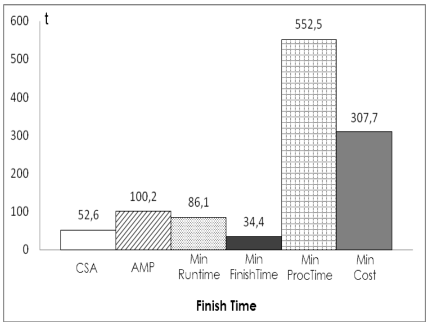


(a) (b)

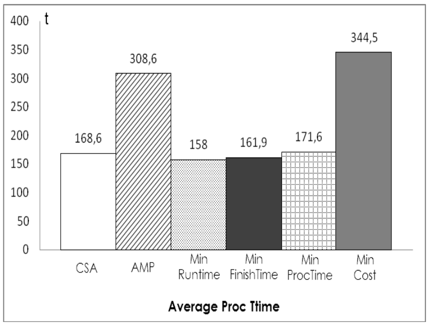
Fig. 2. Average start time (a) and runtime (b)

The minimum *average finish time* (Fig. 3 (a)) was provided by the *MinFinish* algorithm – 34.4. *CSA* has the closest resulting finish time of 52.6 that is 52.9% later. The relative closeness of these values comes from the fact that other related algorithms did not intend to minimize a finish time value and were selecting windows without taking it into account. At the same time *CSA* is picking the most effective alternative among 57 (on the average) allocated at the scheduling cycle: the optimization was carried out at the selection phase. The late average finish time 307.7 is provided by the *MinCost* scheme. This value can be explained not only with a relatively late average start time (see Fig. 2 (a)), but also with a longer (compared to other approaches) execution runtime (see Fig. 2 (b)) due to the use of less productive resource nodes. The finish time obtained by the simplified *MinProcTime* algorithm was relatively high due to the fact that a random window was selected (without any optimization) at each step of the algorithm, though the search was performed on the whole list of available slots. With such a random selection the most effective window by the processor time criterion was near the end of the scheduling interval.

The *average used processor time* (the sum time length of the used slots) for the algorithms considered is presented in Fig. 3 (b). The minimum value provided by *MinRunTime* is 158 time units. *MinFinish*, *CSA* and *MinProcTime* were able to provide comparable results: 161.9, 168.6 and 171.6 respectively. It is worth mentioning that although the simplified *MinProcTime* scheme does not provide the best value, it is only 2% less effective compared to the common *CSA* scheme, while its working time is orders of magnitude less (Tables 1, 2). The most processor time consuming alternatives were obtained by *AMP* and *MinCost* algorithms. Similarly to the execution runtime value, this can be explained by using a random first fitting window (in case of *AMP*) or by using less expensive, and hence less productive, resource nodes (in case of the *MinCost* algorithm), as nodes with a low performance level require more time to execute the job.



(a)



(b)

Fig. 3. Average finish time (a) and CPU usage time (b)

Finally, let us consider the *total job execution cost* (Fig. 4).

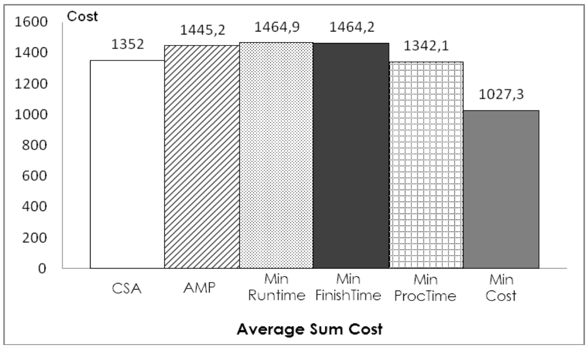


Fig. 4. Average job execution cost

The *MinCost* algorithm has a big advantage over other algorithms presented: it was able to provide the total cost of 1027.3 (note that the total cost limit was set by the user at 1500). Alternatives found with other considered algorithms have approximately the same execution cost. Thus, the cheapest alternatives found by *CSA* have the average total execution cost equal to 1352, that is 31.6% more expensive compared to the result of the *MinCost* scheme, while alternatives found by *MinRunTime* (the most expensive ones) are 42.5% more expensive.

The important factor is a complexity and an actual working time of the algorithms under consideration, especially with the assumption of the algorithm's repeated use during the first stage of the scheduling scheme [6]. In the description of the AEP general scheme it was mentioned that the algorithm has a linear complexity on the number of the available slots. However in has a quadratic complexity with a respect to the number of CPU nodes. Table 1 shows the actual algorithm execution time in milliseconds measured depending on the number of CPU nodes.

The simulation was performed on a regular PC workstation with Intel Core i3 (2 cores @ 2.93 GHz), 3GB RAM on JRE 1.6, and 1000 separate experiments were simulated for each value of the processor nodes numbers {50, 100, 200, 300, 400}. The simulation parameters and assumptions were the same as described in section 3.1, apart from the number of used CPU nodes. A row "*CSA: Alternatives Num*" represents an average number of alternatives found by *CSA* during the single experiment simulation (note that *CSA* is based on multiple runs of *AMP* algorithm). A row "*CSA per Alt*" represents an average working time for the *CSA* algorithm in recalculation for one alternative. The *CSA* scheme has the longest working time that on the average almost reaches 3 seconds when 400 nodes are available.

Table 1. Actual algorithms execution time in ms

CPU nodes number:	50	100	200	300	400
CSA:Alternatives Num	25.9	57	128.4	187.3	252
CSA per Alt	0.33	0.99	3.16	6.79	11.83
CSA	8.5	56.5	405.2	1271	2980.9
AMP	0.3	0.5	1.1	1.6	2.2
MinRunTime	3.2	12	45.5	97.2	169.2
MinFinishTime	3.2	12	45.1	96.9	169
MinProcTime	1.5	5.2	19.4	42.1	74.1
MinCost	1.7	6.3	23.6	52.3	91.5

Besides this time has a near cubic increasing trend with a respect to the nodes number. This trend can be explained by the addition of two following factors: 1) a linear increase of the alternatives number found by *CSA* at each experiment (which makes sense: the linear increase of the available nodes number leads to the proportional increase in the available node processor time; this makes it possible to find (proportionally) more alternatives); 2) a near quadratic complexity of the *AMP* algorithm with a respect to the nodes number, which is used to find single alternatives in *CSA*. Even more complication is added by the need of "cutting" a suitable windows from the list of the available slots [17]. Other considered algorithms will be able to perform a much faster search. The average working time of *MinRunTime*, *MinProcTime* and *MinCost* proves their (at most) quadratic complexity on the number of CPU nodes, and the executions times are suitable for practical use. The *AMP*'s execution time shows even near linear complexity because with a relatively large number of free available resources it was usually able to find a window at the beginning of the scheduling interval (see Fig. 1, a) without the full slot list scan.

Fig. 5 clearly presents the average working duration of the algorithms depending on the number of available CPU nodes (the values were taken from Table 1). (The *CSA* curve is not represented as its working time is incomparably longer than AEP-like algorithms.)

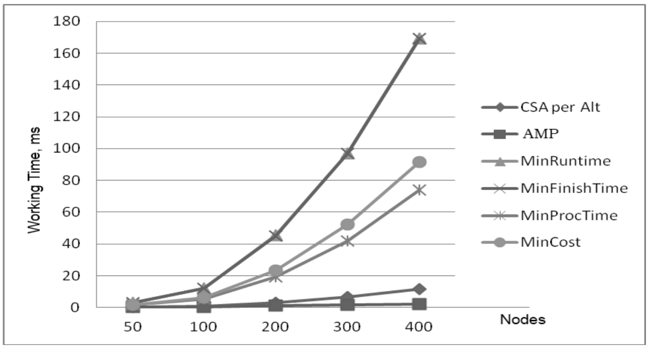


Fig. 5. Average working time duration depending on the available CPU nodes number

Table 2 contains the algorithms working time in milliseconds measured depending on the scheduling interval length.

Table 2. Algorithms working time (in ms) measured depending on the scheduling interval length

Scheduling interval length:	600	1200	1800	2400	3000	3600
Number of slots:	472.6	779.4	1092	1405.1	1718.8	2030.6
CSA:Alternatives Num	57	125.4	196.2	269.8	339.7	412.5
CSA per Alt	0.95	1.91	2.88	3.88	4.87	5.88
CSA	54.2	239.8	565.7	1045.7	1650.5	2424.4
AMP	0.5	0.82	1.1	1.44	1.79	2.14
MinRunTime	11.7	26	40.9	55.5	69.4	84.6
MinFinishTime	11.6	25.7	40.6	55.3	69	84.1
MinProcTime	5	11.1	17.4	23.5	29.5	35.8
MinCost	6.1	13.4	20.9	28.5	35.7	43.5

Overall 1000 single experiments were conducted for each value of the interval length {600, 1200, 1800, 2400, 3000, 3600} and for each considered algorithm an average working time was obtained. The experiment simulation parameters and assumptions were the same as described earlier in this section, apart from the scheduling interval length. A number of CPU nodes was set to 100. Similarly to the previous experiment, *CSA* had the longest working time (about 2.5 seconds with the scheduling interval length equal to 3600 model time units), which is mainly caused by the relatively large number of the formed execution alternatives (on the average more than 400 alternatives on the 3600 interval length). When analyzing the presented values it is easy to see that all proposed algorithms have a linear complexity with the respect to the length of the scheduling interval and, hence, to the number of the available slots (Fig. 6), and their executions times are suitable for on-line scheduling.

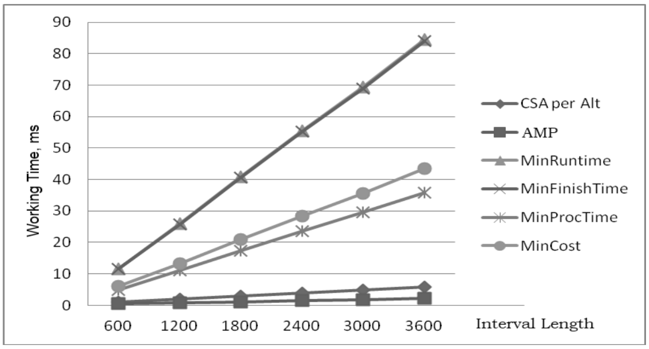


Fig. 6. Average working time duration depending on the scheduling interval length

3.3 Brief Analysis and Related Work

In this work, we propose algorithms for efficient slot selection based on user and administrators defined criteria that feature the *linear* complexity on the number of all available time-slots during the scheduling interval. Besides, in our approach the *job start time* and the *finish time* for slot search algorithms may be considered as criteria specified by users in accordance with the job total allocation cost. It makes an opportunity to perform more flexible scheduling solutions.

The co-allocation algorithm presented in [12] uses the 0-1 IP model with the goal of creating reservation plans satisfying user resource requirements. The number of variables in the proposed algorithm becomes N^3 depending on the number of computer sites N. Thus this approach may be inadequate for an on-line service in practical use. The proposed algorithms have the *quadratic complexity* with a respect to the number of CPU nodes, and not to the number of computer sites as in [12]. A linear IP-driven algorithm is proposed in [2]. It combines the capabilities of IP and genetic algorithm and allows obtaining the best metaschedule that minimises the combined cost of all independent users in a coordinated manner. In [13], the authors propose a MIP model which determines the best scheduling for all the jobs in the queue in environments composed of multiple clusters that act collaboratively. The proposed in [2, 12, 13] scheduling techniques are effective compared with other scheduling techniques under given criteria: the minimum processing cost, the overall makespan, resources utilization etc. However, complexity of the scheduling process is extremely increased by the resources heterogeneity and the co-allocation process, which distributes the tasks of parallel jobs across resource domain boundaries. The degree of complexity may be an obstacle for on-line use in large-scale distributed environments.

As a result it may be stated that each full AEP-based scheme was able to obtain the best result in accordance with the given criterion. This allows to use the proposed algorithms within the whole scheduling scheme [6] at the first stage of the batch job alternatives search. Moreover, each full AEP-based scheme was able to obtain the best result in accordance with the given criterion. Besides, a single run of the AEP-like algorithm had an advantage of 10%-50% over suitable alternatives found with

AMP with a respect to the specified criterion. A directed alternative search at the first stage of the proposed scheduling approach [6, 7] can affect the final distribution and may be favorable for the end users. According to the experimental results, on one hand, the best scheme with top results in start time, finish time, runtime and CPU usage minimization was *MinFinish*. Though in order to obtain such results the algorithm spent almost all user specified budget (1464 of 1500). On the other hand, the *MinCost* scheme was designed precisely to minimize execution expenses and provides 43% advantage over *MinFinish* (1027 of 1500), but the drawback is a more than modest results by other criteria considered. The *MinProcTime* scheme stands apart and represents a class of simplified AEP implementations with a noticeably reduced working time. And though the scheme, compared to other considered algorithms, did not provide any remarkable results, it was on the average only 2% less effective than the *CSA* scheme by the dedicated CPU usage criterion. At the same time its reduced complexity and actual working time allow to use it in a large wide scale distributed environments when other optimization search algorithms prove to be too slow.

4 Conclusions and Future Work

In this work, we address the problem of slot selection and co-allocation for parallel jobs in distributed computing with non-dedicated resources. Each of the AEP algorithms possesses a linear complexity on a total available slots number and a quadratic complexity on a CPU nodes number. The advantage of AEP-based algorithms over the general CSA scheme was shown for each of the considered criteria: start time, finish time, runtime, CPU usage time and total cost.

In our further work, we will refine resource co-allocation algorithms in order to integrate them with scalable co-scheduling strategies [6, 7].

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-316.2012.9), the Russian Foundation for Basic Research (grant no. 12-07-00042), and by the Federal Target Program “Research and scientific-pedagogical cadres of innovative Russia” (state contract no. 16.740.11.0516).

References

1. Lee, Y.C., Wang, C., Zomaya, A.Y., Zhou, B.B.: Profit-driven Scheduling for Cloud Services with Data Access Awareness. J. Parallel and Distributed Computing 72(4), 591–602 (2012)
2. Garg, S.K., Konugurthi, P., Buyya, R.: A Linear Programming-driven Genetic Algorithm for Meta-scheduling on Utility Grids. J. Parallel, Emergent and Distributed Systems 26, 493–517 (2011)

3. Buyya, R., Abramson, D., Giddy, J.: Economic Models for Resource Management and Scheduling in Grid Computing. J. Concurrency and Computation: Practice and Experience 5(14), 1507–1542 (2002)
4. 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)
5. 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 Acad. Publishers (2003)
6. Toporkov, V., Tselishchev, A., Yemelyanov, D., Bobchenkov, A.: Composite Scheduling Strategies in Distributed Computing with Non-dedicated Resources. Procedia Computer Science 9, 176–185 (2012)
7. Toporkov, V., Tselishchev, A., Yemelyanov, D., Bobchenkov, A.: Dependable Strategies for Job-flows Dispatching and Scheduling in Virtual Organizations of Distributed Computing Environments. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) Complex Systems and Dependability. AISC, vol. 170, pp. 289–304. Springer, Heidelberg (2012)
8. Cecchi, M., Capannini, F., Dorigo, A., et al.: The gLite Workload Management System. J. Phys.: Conf. Ser. 219 (6), 062039 (2010)
9. Yu, J., Buyya, R., Ramamohanarao, K.: Workflow Scheduling Algorithms for Grid Computing. In: Xhafa, F., Abraham, A. (eds.) Metaheuristics for Scheduling in Distributed Computing Environments. SCI, vol. 146, pp. 173–214. Springer, Heidelberg (2008)
10. Aida, K., Casanova, H.: Scheduling Mixed-parallel Applications with Advance Reservations. In: 17th IEEE Int. Symposium on HPDC, pp. 65–74. IEEE CS Press, New York (2008)
11. Elmroth, E., Tordsson, J.: A Standards-based Grid Resource Brokering Service Supporting Advance Reservations, Coallocation and Cross-Grid Interoperability. J. Concurrency and Computation: Practice and Experience 25(18), 2298–2335 (2009)
12. Takefusa, A., Nakada, H., Kudoh, T., Tanaka, Y.: An Advance Reservation-based Co-allocation Algorithm for Distributed Computers and Network Bandwidth on QoS-guaranteed Grids. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2010. LNCS, vol. 6253, pp. 16–34. Springer, Heidelberg (2010)
13. Blanco, H., Guirado, F., Lérida, J.L., Albornoz, V.M.: MIP Model Scheduling for Multi-clusters. In: Caragiannis, I., et al. (eds.) Euro-Par Workshops 2012. LNCS, vol. 7640, pp. 196–206. Springer, Heidelberg (2013)
14. Moab Adaptive Computing Suite, `http://www.adaptivecomputing.com`
15. Toporkov, V., Toporkova, A., Bobchenkov, A., Yemelyanov, D.: Resource Selection Algorithms for Economic Scheduling in Distributed Systems. Procedia Computer Science 4, 2267–2276 (2011)
16. Toporkov, V., Yemelyanov, D., Toporkova, A., Bobchenkov, A.: Resource Co-allocation Algorithms for Job Batch Scheduling in Dependable Distributed Computing. In: Zamojski, W., Kacprzyk, J., Mazurkiewicz, J., Sugier, J., Walkowiak, T. (eds.) Dependable Computer Systems. AICS, vol. 97, pp. 243–256. Springer, Heidelberg (2011)
17. Toporkov, V., Bobchenkov, A., Toporkova, A., Tselishchev, A., Yemelyanov, D.: Slot Selection and Co-allocation for Economic Scheduling in Distributed Computing. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 368–383. Springer, Heidelberg (2011)

Determination of Dependence of Performance from Specifications of Separate Components of the Hybrid Personal Computing System Based on GPU-Processors

Daulet Akhmedov[1], Suleimen Yelubayev[1], Timur Bopeyev[1], Farida Abdoldina[2], Daulet Muratov[1], Rustam Povetkin[1], and Aibek Karataev[1]

[1] Laboratory of Space Systems Simulation, The Institute of Space Technique and Technologies, Almaty, Republic of Kazakhstan
{akhmedov.d,elubaev.s,bopeyev.t}@istt.kz
[2] Department of Software of Systems and Networks, The Institute of Information and Telecommunication Technologies, Kazakh National Technical University Named after K.I.Satpaev,
Almaty, Republic of Kazakhstan
farida_mail@mail.ru

Abstract. In this paper selection of electronic component base as one of issues of development of personal hybrid computing system is reviewed. Test results of performance of experimental prototype of personal hybrid computing system based on 3 NVidia Tesla C2050 graphic processors have been explained. Row of experiments for evaluating system performance has been carried out with different RAM clock frequency (1066 and 1333MHz) and memory volume (from 2 up to 24 GB with 2GB step), also the dependence of performance from bandwidth of PCI-Express x8 and PCI-Express x16 slots has been studied.

Results of this research laid down in a basis of development process of experimental prototype of the personal hybrid computing system.

Keywords: Parallel computing, personal hybrid computing systems, graphics processors, CUDA technology.

1 Introduction

In this paper results of the researches accomplished within the confines of the scientific project "Develop the personal hybrid computing system based on GPU-processors" which is entering into state-financed program "Develop technology of creation of supercomputer hybrid cluster with application of GPU-processors" of the Ministry of Education and Science of Republic of Kazakhstan have been presented. The purpose of project is to carry out developmental works to create personal hybrid computing system.

When designing of personal hybrid computing system main complexity is the calculations of circuit of optimum ratio of a powerful CPU and a few graphic processors, cooling system, optimal power scheme. This paper presents the results of selection of system's electronic component base.

Within the confines of project the experimental prototype of personal hybrid computing system is created. For this purpose at a preliminary stage the experimental prototype of personal hybrid computing system has been created and on its basis researches of dependence of actual performance from specifications of separate components of system hardware were accomplished.

Obtained results were allowed to evaluate impact level of separate components on performance of personal hybrid computing system and to make sure of regularity of chosen direction of creation of personal hybrid computing system, also results laid down in a basis of creation process of a experimental prototype of personal hybrid computing system.

Results of this research will allow to solve a task of optimal characteristics selection of hardware for achievement of the best price/productivity ratio of created system.

2 Personal Hybrid Computing System

Experimental prototype of personal hybrid computing system based on graphic cards with use of CUDA-technology [1] has been developed. Experimental prototype of personal hybrid computing system has peak performance for about 3 TFlops single accuracy and 1.5 TFlops double accuracy and allows to replace a small cluster by itself.

Power consumption of personal hybrid computing system is about 1200W. Performance ratio on Watt of power consumption is 552.5 MFlops/Watt.

Performance of personal hybrid computing system per cost of possession is also extremely high. Cost of experimental prototype of PHCS is $13.4 thousand. Price/performance ratio is about $20 for 1 GFlops/sec on Linpack test.

3 Testing. Analysis Results

While evaluating of performance of clusters and supercomputers 2 variants are used: peak performance - theoretical limit of performance of these processors and actual performance, which is reached by cluster or computer while solving practical tasks. For testing of hybrid computing systems based on graphic processors Cuda Accelerated Linpack [2] are used.

Performance testing of experimental prototype of personal hybrid computing systems based on 3 graphic processors has been carried out with different RAM clock frequency (1066 and 1333MHz) and memory volume (from 2 up to 24 GB with 2GB step), also the dependence of performance from bandwidth of PCI-Express has been studied.

At the time of testing of performance of experimental prototype of personal hybrid computing system the configuration consisted of: processor Intel Core i7-960, motherboard - Asus Rampage III, video card GeForce GTX9800+, GPU NVidia Tesla C2050, RAM 1066/1333 MHz. Software: OS Linux Ubuntu 11.04 (2.6.38), CUDA version 4 .2, math library Intel MKL 10, MPI library release - OpenMPI 1.4.

With a clock frequency of RAM 1066 MHz and with memory volume in 2 GB up to 24 GB system performance grows up to 2.5 times. Values of system performance in dependence of RAM and memory volume presented in Table 1.

Table 1. Testing results of experimental prototype of personal hybrid computing system based on 3 graphic processors Nvidia Tesla C2050 at various RAM volume and clock frequency 1066 MHz, 1333 MHz

Test number	RAM volume, GB	Matrix size, N	Performance with RAM clock frequency - 1066 MHz, GFlops	Performance with RAM clock frequency - 1333 MHz, GFlops
1	2	14273	227,8	259.0
2	4	20822	327,2	350.4
4	8	29447	450,3	479.0
8	16	41645	539,1	605.5
12	24	52224	622,6	663.4

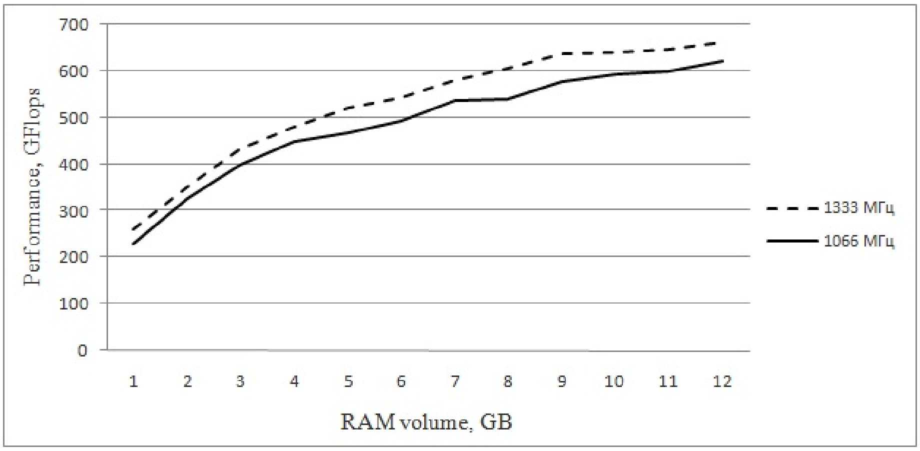


Fig. 1. Dependence of performance of experimental prototype of personal hybrid computing system with 3 GPU-processors from RAM clock frequency

Analyzing changes of actual performance of experimental prototype with various clock frequencies of memory, average value of performance increases for 9.03% or 42.99 GFlops while increasing clock frequency for 25 % from 1066 up to 1366 MHz have been received (Fig. 1).

Modern CPUs supports only40 PCI-Express lines that limits quantity of connected PCI-Express devices. In our case two connection options via x8 and x16 buses are reviewed.

Analyzing changes of actual performance of experimental prototype with different PCI-Express bus rate, average value of performance increase for 3,11% or 14,46 GFlops has been received (Table 2).

It's possible to make a conclusion: PCI-Express bus rate hasn't sizable impact on performance of personal hybrid computing system.

Table 2. Testing results of performance dependence of experimental prototype of personal hybrid computing system from PCI-Express bus bandwidth

Test number	RAM volume, GB	Matrix size, N	Performance with PCI Express x16, GFlops	Performance with PCI Express x8, GFlops
1	2	14273	253,0	262,2
2	4	20822	358,4	333,6
4	8	29447	443,7	425,2
8	16	41645	515,4	506,6
12	24	52224	555,8	536,0

4 Conclusion

Conducted analysis of evaluating results of performance of experimental prototype of personal hybrid computing system has demonstrated that actual performance has direct dependence from system's RAM volume. Given results of scientific experiments show that with increase of RAM volume performance of system increases non-linearly. In turn increase of graphic processors amount demands increase of RAM volume for effective loading of graphic processors and for obtaining the best system performance.

Also it was defined that increase of RAM frequency at bigger amount of graphic processors allows to receive a bigger gain of actual performance in Linpack test - 4,7% for 1-2 graphic processors and 9% for 3 graphic processors.

Tests of experimental prototype of personal hybrid computing system allow to determine actual system performance in Linpack test, which was 663 GFlops - 44.2% of peak performance. This value isn't a limit as actual performance can reach up to 70% of peak performance at best specifications of main components of system, higher result can be reached.

During preparation of materials of this paper new results were received. For personal hybrid computing system in FullTower form-factor based on 2 CPUs Intel Xeon E5-2690 2.9 GHz and 4 GPUs NVidia Tesla C2075 it was succeeded to reach 61,7% of performance in the Linpack test.

References

1. Boreskov A.V., Kharlamov A.A.: Osnovy raboty s tekhnologiey CUDA (Basic work with the CUDA technology). Moscow (2010) (in Russian)
2. NVidia Technical support, `http://developer.nvidia.com`

Design of Distributed Parallel Computing Using by MapReduce/MPI Technology

Darkhan Akhmed-Zaki, Nargozy Danaev,
Bazargul Matkerim, and Amanzhol Bektemessov

Al-Farabi Kazakh National University,
Al-Farabi. 71, 050040 Almaty, Kazakhstan
{Darhan.Ahmed-Zaki,Nargozy.Danaev}@kaznu.kz
http://www.kaznu.kz

Abstract. This paper describes a constructive approach of distributed parallel computing using by hybrid union of MAPREDUCE and MPI technologies for solving oil extracting problems. We extend a common architecture of MAPREDUCE model by organizing decomposition of computational domain at different stages of MAPREDUCE process. We describes Model Driven Architecture (MDA) models for developing formal views of high-performance computing technologies using MAPREDUCE. We made computing experiments and show on specific HPC infrastructure. All implementations of programs is realize on Java platform. This approach will possible one of the ways to do cloud computing on high performance heterogeneous systems.

Keywords: MAPREDUCE, MPI, mpiJava, distributed parallel computing on heterogeneous systems.

1 Introduction

The basic principles of organizing parallel and distributed computing are known for a long time [1,2]. The majority of modern researches is focused on researching new methods of developing the effective parallel computing for large-scale problems [3, 4]. Mostly actual problems are in parallel computing on highly heterogeneous (hybrid) systems, developing parallel program designing systems - frame solution (”skeleton”), parallel program’s code verification and reliability of developed systems [5-7]. One of the popular problems now is find answer of next question ”Can we effectively integrate cloud (virtual) technology with parallel computing technology?”. It is more actual in organization of reliable distributed (on heterogeneous computing resources) high-performance processing of large volumes of heterogeneous data[8]. Due to the high level of practical relevance, a lot of works [9-11] are devoted to combining the advantages of parallel, distributed and cloud computing technologies. In our work we describe constructive approach for hybrid combining of MAPREDUCE [12] and MPI [13] technology for realize distributed parallel computing on heterogeneous systems to solve one problem of oil production. General description of MAPREDUCE technology is presented in many papers [14, 15]. The main problem in this field is to provide an effective load balancing on existing resources and high reliability of the distributed data processing systems. MAPREDUCE technology is a clear separate roles of computing nodes to "mapper" and "reducer". The disadvantage is the impossibility of providing communications between nodes during the calculation process, such that each node carries out its own task and only at the end of map-reduce operations to combine calculated data. Due to complexity of the computational domain structures (anisotropic and inhomogeneous porous medium) we choose the problem of computing oil-field's pressure for 3D case of high-performance data processing and visualization. We develop our parallel computing on mpiJava [16], such that MAPREDUCE technology is also implemented using Java and easily integrated with virtual platforms on Java Virtual Machine (JVM). Lastest is provided a cross-platform application for our case. This article consist on follow sections: mathematical model of pressure fields distributions in 3D anisotropic porous media and creating parallel computing algorithm (in MPI), build its discrete models (on Model Driven Architecture - MDA), realize experimental computing and discussion of results.

2 Mathematical Model

Oil Production problem. Suppose 3D model problem of the fluid flows in porous elastic anisotropic medium in a cube $\Omega = [0,T] \times K\{0 \leq x \leq 1, 0 \leq y \leq 1, 0 \leq z \leq 1\}$ [17]:

$$\frac{\partial P}{\partial t} = \frac{\partial}{\partial x}(\phi(x,y,z)\frac{\partial P}{\partial x}) + \frac{\partial}{\partial y}(\phi(x,y,z)\frac{\partial P}{\partial y}) + \frac{\partial}{\partial z}(\phi(x,y,z)\frac{\partial P}{\partial z}) + f(x,y,z), \quad (1)$$

The initial condition:

$$P(0,x,y,z) = \varphi(x,y,z). \quad (2)$$

Boundary conditions:

$$\left.\frac{\partial P}{\partial n}\right|_{\Gamma} = 0, \quad (3)$$

where Γ is sides of Ω cube. There, $P(x,y,z,t)$ – reservoir pressure at point (x,y,z) and time t; $\phi(x,y,z)$ – formation pressure conductivity factor; $f(x,y,z)$ – density of the sources, wells impact; $\varphi(x,y,z)$ – initial means of deposit pressure.

We use an explicit iterative method - Jacobi method [18] to solve the problem (1) – (3). Parallel computing algorithm for solving the problem (1) – (3) is based on the known approach of three-dimensional decomposition of computation domains (3D cube view of oil-fields) with inter-exchanging of boundary values, as shown in Figure 1. In first step, basic domain divided to layers, after each layer divide into cubes and organize calculation with closed to computational nodes. At next, we provides communication and inter-exchanging of boundary values in each "micro-cube".

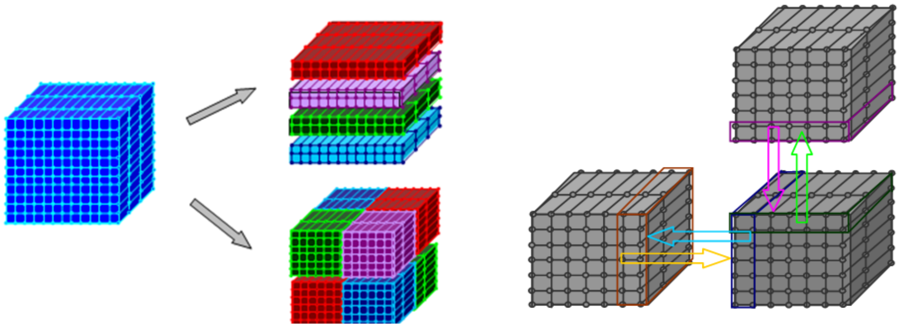


Fig. 1. Three-dimensional decomposition of domain and interchange of boundary values

3 MAPREDUCE/MPI Platform

In order to organize distributed parallel computing, we choose one of the commonly used platforms MAPREDUCE - HADOOP [19]. In general, high-performance processing is usually used for commercial purposes - a distributed file system Google (GFS) with support of the MapReduce implementation, as well as the open-source implementation of GFS - HDFS (Hadoop distributed file system) [20]. Hadoop cluster is a collection of different characteristics of computing nodes - computers connected by a network for co-processing of large volumes of data distributed across the cluster. The main idea underlying the HDFS is to divide a particular user's data blocks and replication of data blocks to local disks of nodes in the cluster. The general principle of system uses a master / slave (main / sub) architecture where the primary node maintains information about the file namespace (metadata, directory structure, the correspondence between the specified files and data blocks, the location of data blocks and permissions), and slave nodes directly control data blocks, i.e. processing (computing) of the data. MapReduce applications operate with the list of key-value pairs as input and consist of two main methods of Map and Reduce. The Map method treats each pair of "key-value" in the input list separately and generates outputs of one or more pairs of "key-value" as a result [12]:

$$\text{map(key, value)} \Rightarrow [\text{(key, value)}]. \tag{4}$$

The Reduce method aggregates the output of Map method. It gets the key and the list of values assigned to this key as input, performs user-defined operations on them and outputs one or more key-value pairs:

$$\text{reduce(key, [value])} \Rightarrow [\text{(key, value)}]. \tag{5}$$

This process is illustrated in Figures 2 and 3 in the case for (1) - (3), where computational domain is decomposed into 8 blocks is distributed (MapReduce level) and parallel (Map Level 2) computing on MPI. Exactly between stages 3

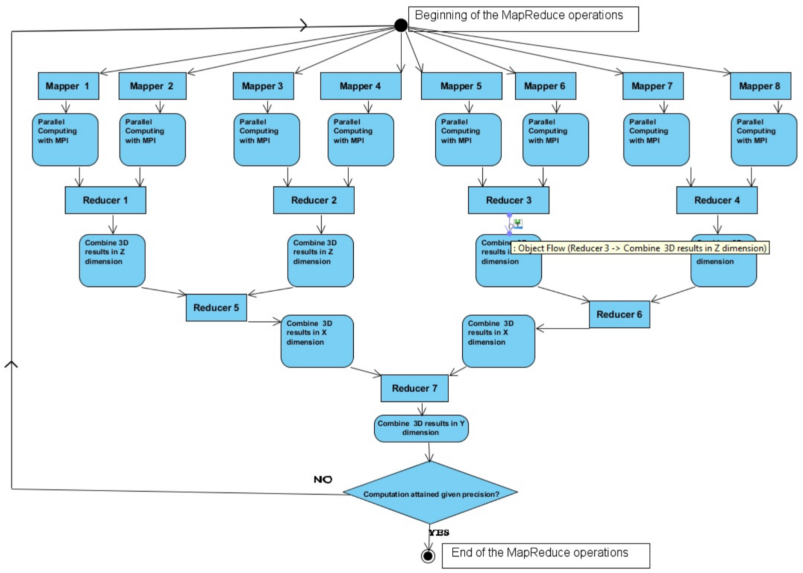

Beginning of the MapReduce operations
Mapper 1
Mapper 2
Mapper 3
Mapper 4
Mapper 5
Mapper 6
Mapper 7
Mapper 8
Parallel Computing with MPI
Reducer 1
Reducer 2
Reducer 3
Reducer 4
Combine 3D results in Z dimension
: Object Flow (Reducer 3 -> Combine 3D results in Z dimension)
Reducer 5
Reducer 6
Combine 3D results in X dimension
Reducer 7
Combine 3D results in Y dimension
Computation attained given precision?
NO
YES
End of the MapReduce operations

Fig. 2. The sequence of MapReduce calculations of oil extraction problem

and 4 on Map level 2 MPI computations is accomplished, which is integrated with MPI computations in each map node. On the stage of doing reduce tasks we used common process - inter-exchanging of boundary values into divided (as a result of decomposition) computational domain. According to the classification of [21], our case is the implementation of a parallel algorithm on MapReduce model corresponds to the common third classes - "algorithms where the content of one iteration is represented as an execution of a single MapReduce model". Thus, if the input initial data of (1) - (3) are as "macro-cube" with $(N * N * N)$ dimension, then after the first stage of decomposition get "micro-cubs" (Map level 1) of dimension $(N/2 * N/2 * N/2)$. Further, use MPI technology with the second stage of decomposition - "micro-cube" (Map level 2) of dimension $(N/4 * N/4 * N/4)$. Then we will have:

- **Map:**
  - Compute Proc_3D_MPI values of "micro-cube" in select nodes and assign it as a new object.
  - Input: (cluster id, object_old_1). (6)
  - Output: (cluster id, object_new_1).
- **Reduce:**
  - Combine results of calculation, exchange borders values and assign it as a new object.
  - Input: (cluster id, (list of all objects_new_1 from nodes)). (7)
  - Output: (cluster id, object_new_2).

Here *Proc_3D_MPI* - implements calculation of 3D problem on data "micro-cube" in the selected compute node; *object_old_1*-input data "micro-cube" is the union of three-dimensional matrices $P_{old\_ijk}, \phi_{old\_ijk}, f_{ijk}$; *object_ new_1* – output of calculated data "micro-cube" – P_{new_ijk}; *object_ new_2*— output of calculated data "micro-cube" with exchange of boundary data - the union adjacent pairs of three-dimensional matrices on the axes with the dimension, such as on Z axis - $(N/4 * N/4 * N/2)$.

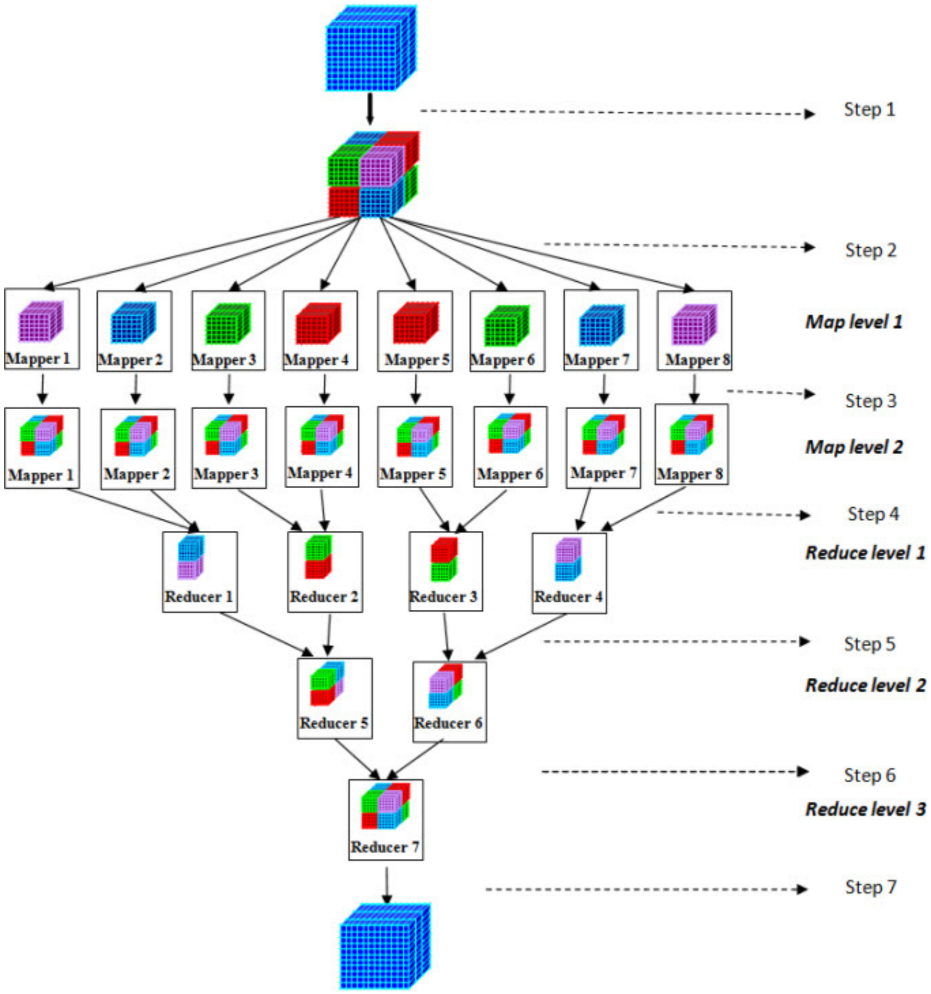


Fig. 3. Computational domain decomposition at different stages and levels of MapReduce jobs

4 Discrete Models of Distributed Parallel Computations on MDA

Software development based on model - Model Driven Architecture (MDA) is a paradigm in which a code writing stage: partially or fully automated [22]. Discrete model development on MDA approach based on creation an appropriate extension of UML 2.0 language [23] for a specific area of formal modeling - for example, distributed parallel computing problem of oil production. Moreover, a formal description of interfaces on MDA allows to generating parallel programs - making framework ("skeletons") [24]. Main tools of MDA are the platform independent model (PIM) and a model that takes into account the specific features of the platform (Platform Specific Model - PSM) [22]. Below we propose an MDA model - Sequence of load balancing between nodes in given architecture, shown in Figure 4. All designed diagrams after using ATL [23] and our framework generate partially numerical parallel code in java classes for mapper (reducer) nodes.

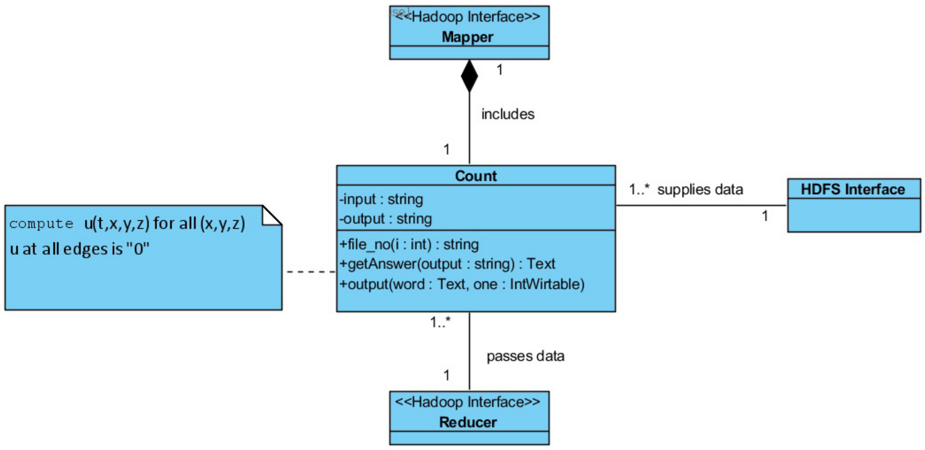


Fig. 4. Sequence of load balancing between nodes in given architecture

5 Computational Experiment

We perform numerical experiments in next infrastructure: hierarchically arranged Hadoop platform. This architecture includes 8 PCs with IntelCorei7 processors and 4 GB RAM, 1 server with 8 cores, HPC "URSA" cluster - 128 cores. Topology of this Hadoop system as follows:

1. Main node of cluster (master node).
2. 8, 64, 125 computing nodes (slave nodes).

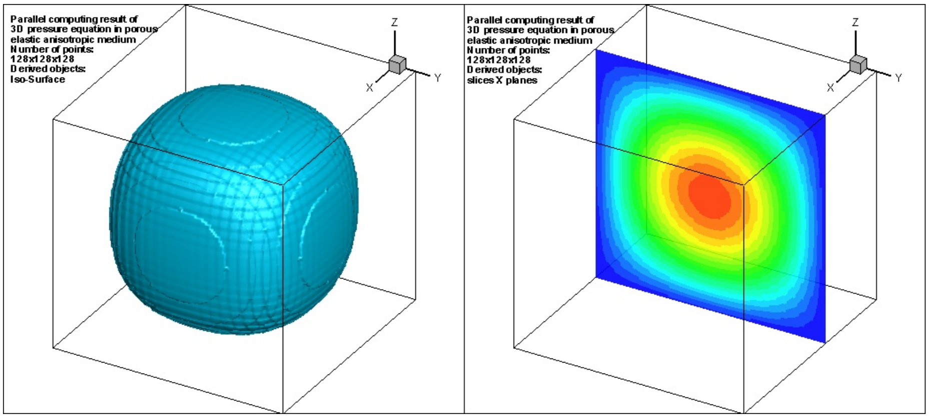

Parallel computing result of
3D pressure equation in porous
elastic anisotropic medium
Number of points:
128x128x128
Derived objects:
Iso-Surface
Parallel computing result of
3D pressure equation in porous
elastic anisotropic medium
Number of points:
128x128x128
Derived objects:
slices X planes
X
Y
Z

Fig. 5. Result of 3D decomposition based parallel program

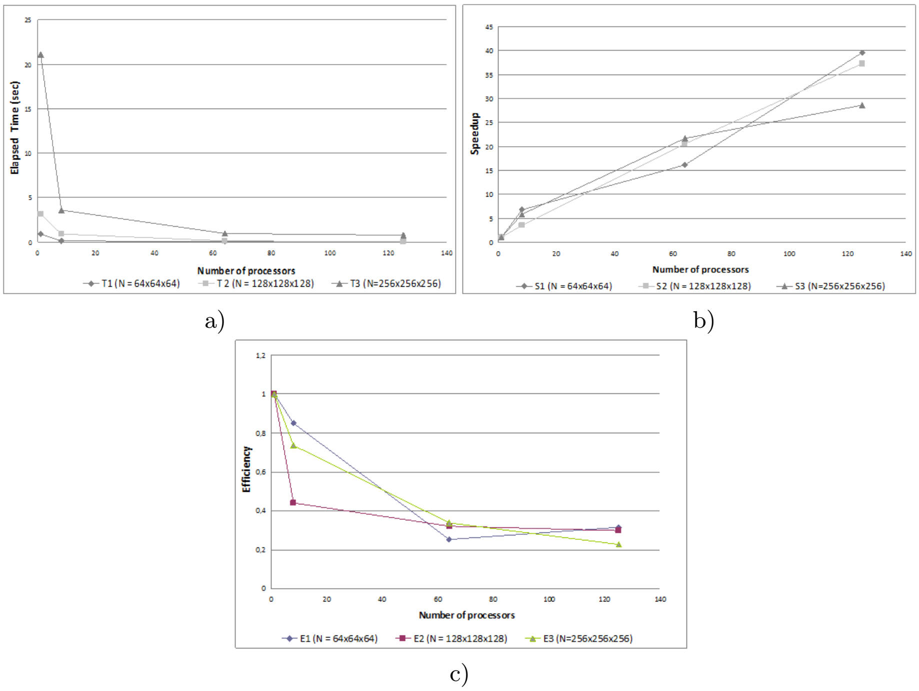


Fig. 6. Time (a), speedup (b) and efficiency (c) for MapReduce algorithm on nodes

The master node of cluster can communicate over the network with other nodes in Hadoop cluster and also remotely control operational systems of slave nodes we installed SSH on every node. Next step in a cluster setting up process we have generated the RSA keys pair on master node of cluster. These keys are needed to establish a secure channel through which cluster nodes could communicate. The private key is stored on master node and public key should be distributed among other remaining cluster nodes. For key generation we used SSH protocol commands. Next is run problem (1) - (3) finding the values of pressure field distributions in 3D anisotropic porous medium with various regimes of oil production with parallel computing algorithm in $N = 128*128*128$. Figure 5 shows solutions obtained in the Iso surface, which demonstrate the overall convergence of parallel algorithm. To evaluate the effectiveness of distributed parallel computing algorithm for solving 3D problems is tested for 8, 64 and 125 nodes of Hadoop cluster (including "URSA" nodes) in the following cases (points count): 1) $N = 64 * 64 * 64$, 2) $N = 128 * 128 * 128$, 3) $N = 256 * 256 * 256$.

Discussion. Results of computation time, speedup and efficiency are shown in Figure 6. As you can see on figures, the proposed system realize distributed parallel computing algorithm quicker than sequence program in one node. Additionally for our case we observe if number of points is increased then influence and calculation time of data communications between nodes is also increased. In case when speedup is down, this requires adding new computational nodes or needed using more effective load balancing algorithms. We see MapReduce is not fully provide to realize the good features of MPI. The important features of MapReduce technology is its ability to handle faults transparently [21]. But during verification is very hard to check a collective operation on all nodes. Using MPI with MapReduce algorithms demonstrate basic limitations in the processing of intermediate data of communications. In our experiments we get outputs - preferred using nonblocking collective operations, which can provided a speedup of up to 15 percent over than the blocking operations. This support to design other modern programming and parallelization technologies or ideas in future.

6 Related Work

Closest to our research paper is of S. Srirama et al. [21], where he studies the creation of scientific cloud computing (SciCloud) addressed to problem of organization of iterative algorithms for MapReduce model. In particular, in his paper the classification of such algorithms with its adapting into MapReduce model is presented and some examples of problems implementation (methods of conjugate gradient, clustering, etc.) with performance analysis on Hadoop by comparison with Twister are shown. This paper noted that Twister is best suited to specific kinds of iterative algorithms on MapReduce, but MapReduce model implementation on Hadoop have better fault tolerance features than Twister [25]. Other applications of adapting of MapReduce model to parallel computation are presented in [26-27]. But, in general, the problem of efficient organization of iterative calculations on MapReduce model remains, particularly, the problems of algorithms scalability and their adaptation for wide classes of scientific problems are still open. Also there is no clear approach that guarantees reliability of such complex systems. Our proposed constructive approach of hybrid combining technologies allow to organize distributed parallel computing on heterogeneous systems solutions for 3D modeling of chosen oil production problem, which is much more complicated than discussed problems in [25-27].

7 Summary

As the results of research we consider constructive approach for distributed parallel computing using MAPREDUCE and MPI:

– designed architecture of MapReduce model – compute problem (1)-(3);
– showed the structure of Map (6) Reduce (7) methods implementation and realization of computational domain decomposition at different stages and levels of MapReduce models;
– described of MDA models - high-performance data processing on MapReduce compute nodes;
– made computational experiment conducted on chosen specific infrastructure.

Also, partially, results of this research is used in designing web distributed information system for analysis and development of oil and gas fields. This system supports visualization of 3D hydrodynamic calculation of oil deposits processes and allow to work in framework which use distributed parallel computing on MAPREDUCE/MPI. In general, one the results of this paper is design the architecture that implemented on benefits of MAPREDUCE technology using Java and organizing parallel computing on mpiJava. This approach is one of the ways to organize cloud computing on high performance heterogeneous systems.

References

1. Gelenbe, E., Lichnewsky, A., Staphylopatis, A.: Experience with the parallel solution of partial-differential equations on a distributed computing system. IEEE Transactions on Computers 31(12), 1157–1164 (1982)
2. Gropp, W., Lusk, E., Doss, N., et al.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22(6), 789–828 (1996)
3. Sunderam, V.S., Geist, G.A., Dongarra, J., et al.: The PVM concurrent computing system - evolution, experiences, and trends. Parallel Computing 20(4), 531–545 (1994)
4. Malyshkin, V.: Assembling of Parallel Programs for Large Scale Numerical Modeling, p. 1021. IGI Global, Chicago (2010)
5. Becker, J.C., Dagum, L.: Particle simulation on heterogeneous distributed supercomputers. Concurrency-Practice and Experience 5(4), 367–377 (1993)
6. Fougère, D., Gorodnichev, M., Malyshkin, N., Malyshkin, V., Merkulov, A., Roux, B.: NumGrid middleware: MPI support for computational grids. In: Malyshkin, V.E. (ed.) PaCT 2005. LNCS, vol. 3606, pp. 313–320. Springer, Heidelberg (2005)

7. Diaz, J., Munoz-Caro, C., Nino, A.A.: Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. IEEE Transactions on Parallel and Distributed Systems 23(8), 1369–1386 (2012)
8. Wang, J., Liu, Z.: Parallel Data Mining Optimal Algorithm of Virtual Cluster. In: International Conference on Fuzzy Systems and Knowledge, vol. 5, pp. 358–362 (2008)
9. Pandey, S., Buyya, R.: Scheduling Workflow Applications Based on Multi-source Parallel Data Retrieval in Distributed Computing Networks. Computer J. 55(11), 1288–1308 (2012)
10. Liu, H., Orban, D.: GridBatch: Cloud Computing for Large-Scale Data-Intensive Batch Applications. In: CCGRID 2008, vol. 1, pp. 295–305 (2008)
11. Valilai, O.F., Houshmand, M.: A collaborative and integrated platform to support distributed manufacturing system using a service-oriented approach based on cloud computing paradigm. Robotics and Computer-Integrated Manufacturing 29(1), 110–127 (2013)
12. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
13. Fagg, G.E., Dongarra, J.: FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In: Dongarra, J., Kacsuk, P., Podhorszki, N. (eds.) PVM/MPI 2000. LNCS, vol. 1908, pp. 346–353. Springer, Heidelberg (2000)
14. He, Q., Wang, Q., Zhuang, F., Tan, Q., Shi, Z.: Parallel CLARANS Clustering Based on MapReduce. Energy Procedia 13, 3269–3279 (2011)
15. Cohen, J.: Graph twiddling in a MapReduce world. Computing in Science and Engineering 11, 29–41 (2009)
16. Baker, M.: MpiJava: A Java interface to MPI. University of Portsmouth, Portsmouth (2010)
17. Aziz, H., Sattar, A.: Mathematical modeling of reservoir systems. Nedra, Moscow (1982)
18. Demmel, J., Veselic, K.: Jacobis method is more accurate than QR. SIAM Journal on Matrix Analysis and Applications 13(4), 1204–1245 (1992)
19. Lam, C.: Hadoop In Action. Manning Publications Co., Stamford (2010)
20. Apache Software Foundation, HDFS (2011), `http://hadoop.apache.org/common/docs/current/hdfs_design.html`
21. Srirama, S.N., Jakovits, P., Vainikko, E.: Adapting scientific computing problems to clouds using MapReduce. Future Generation Computer Systems 28(2), 184–192 (2012)
22. Frankel, D.: Model Driven Architecture. Applying MDA to Enterprise Computing. Wiley Publishing, Indiana (2003)
23. Lugato, D.: Model-driven engineering for high-performance computing applications. In: The 19th IASTED International Conference on Modelling and Simulation, pp. 18–33. IEEE Press, New York (2008)
24. Klusik, U., Loogen, R., Priebe, S., Rubio, F.: Implementation skeletons in Eden: Low-effort parallel programming. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 71–88. Springer, Heidelberg (2001)
25. Srirama, S.N., Batrashev, O., Jakovits, P., et al.: Scalability of parallel scientific applications on the cloud. Scientific Programming 19(2-3), 91–105 (2011)
26. Plimpton, S., Devine, K.: MapReduce in MPI for Large-scale graph algorithms. Parallel Computing 37(9), 610–632 (2011)
27. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative MapReduce. In: HPDC 2010, pp. 810–818. ACM, New York (2010)

PowerVisor: A Toolset for Visualizing Energy Consumption and Heat Dissipation Processes in Modern Processor Architectures

Alexey Baranov[1], Peter Panfilov[1], and Dmitry Ponomarev[2]

[1] National Research University – Higher School of Economics, Department of Computer Systems and Networks, B.Trekhsvyatitelski per. 3, Moscow 109028
aleksej_baranov@pochta.ru, panfilov@miem.edu.ru
[2] State University of New York at Binghamton, Department of Computer Science, P.O. Box 6000, Binghamton, NY 13902-6000
dima@cs.binghamton.edu

Abstract. In this paper, we present PowerVisor – a new toolset for visualizing the energy consumption and heat dissipation processes in multicore and multithreaded processors. PowerVisor uses the data on energy consumption and heat dissipation in processor's units generated by the execution-driven processor simulator. It graphically depicts the energy consumption and heat dissipation dynamics for the applications that use the processor resources such as caches, cores and interconnects. We present the implementation of PowerVisor and describe how it can be used in research and computer architecture eduication.

1 Introduction

Microprocessor development efforts and related research in the area of computer architecture have always relied on the use of accurate functional and timing performance simulators. The high complexity of such projects makes it a challenging task of building accurate forecasts of processor performance using analytical models, which makes simulation the only practical alternative. The use of simulation models reduces cost and development time of new hardware solutions by studying and testing them on software models first, before implementing in silicon.

Modern computer architectures require that processor designers carefully address problems associated with the power consumption and heat dissipation within the processor. The increase in clock speed, high transistor integration densities, higher number of transistors on a chip, and the transition to multiple cores all lead to an increase in power consumption and heat dissipation within a microprocessor chip. This requires the development of new methods and engineering solutions to address the heat dissipation in multi- and many-core designs, which in turn necessitates the development and use of software tools to model and monitor the processes of energy consumption and heat dissipation[1-4].

2 Modern Processors: Architecture and Simulation

Power and energy have become first-order design constraints for modern microprocessor systems. There are many factors in today's processor architecture design that drive the need for more energy-aware solutions. Some of these solutions include the following:

- Dynamic adaptation of processor resources to match the demands of executing applications and improve power/performance efficiency.
- Improving technology to reduce energy consumption, deployment of the new materials to reduce the leakage current, lowering the CPU core voltage;
- Integration of the on-chip temperature sensors and thermal protection system, which triggers adequate system response after encountering temperature emergencies;
- The emergence of dynamic voltage/frequency scaling techniques, including techniques at the granularity of smaller clock domains;
- The emergence of energy-saving modes to put the processor into a "sleep" mode at low loads [5,6].

A large number of cycle-accurate processor simulators exist in the public domain, they model both the processor performance, and also its power and energy consumption. However, all those tools lack the visualization engine to understand the dynamics of the power dissipation in the course of program execution. To fill this void, we developed the PowerVisor toolset on top of M-Sim simulator [7]. M-Sim (publicly available at http://www.cs.binghamton.edu/~msim) is an extension of a widely-used Simplescalar simulator (http://www.simplescalar.com) that supports simultaneous multithreading within each processor core and also supports the simulation of multicore architectures. Currently, it supports the execution of Alpha EV6 run SPEC 2006 benchmarks to completion. During every simulation cycle, a significant amount of execution statistics is computed. In the next section, we explain how PowerVisor was implemented on top of M-Sim.

3 PowerVisor Implementation

In our implementation of a PowerVisor addition to M-Sim, we use *offline approach* to the interaction with the external processor simulation modules. 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 visualize engine, which depicts the results dynamically in a graphical form. 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. However, the offline approach requires minimal synchronization between the components, and is therefore easier to implement. The use of the 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. It makes possible to observe the level of power consumption and heat dissipation at any time. The main problem with this approach is the need to store the trace file of a large volume.

The current version of PowerVisor was designed using the offline approach. In this design, the entire visualization system is separated into two larger parts: power statistics collection module and standalone visualizer. The general architecture of PowerVisor implemented using offline approach is shown in Figure 1.

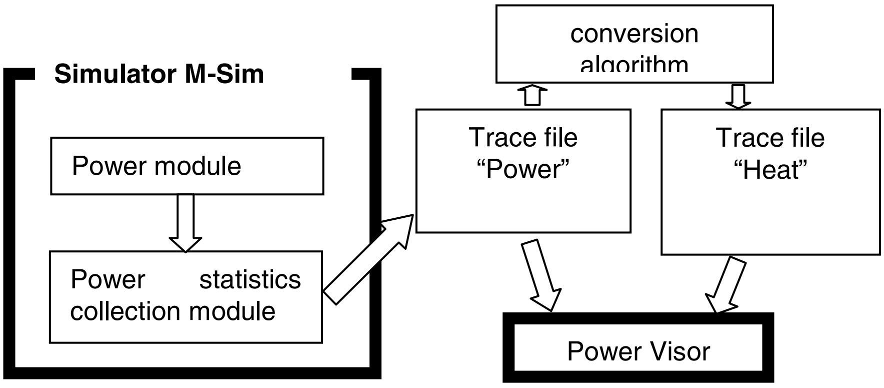


Fig. 1. Interaction of PowerVisor and M-Sim using the trace file

Power module is an integral part of the M-Sim simulator, particularly responsible for the simulation of energy consumption in the processor units.

All events in the simulator, like load / store memory operations, data transfers via system bus or interconnection network, cache accesses, or basic computational operations are associated with the correspondent constant energy values derived from the parameterized power models.

The Power statistics collection module is built into the M-Sim simulator. It tracks the events in the simulator, stores simulation data in a special trace file, and relays collected energy data to PowerVisor. In contrast to the basic data acquisition module in M-Sim simulator, the modified Power statistics collection module provides information about events in a cycle-accurate fashion.

The Power trace file contains information (in the text form) about power consumption in various processor blocks. To limit the size of trace file, the maximum number of cycles is defined for data collection on simulation events. Also an option is provided of simulation data selection for inclusion into Power trace file (e.g., energy consumption data for the cache unit only, or the system bus only, or the entire processor core, etc.).

The Conversion algorithm module transforms the simulation trace with energy consumption data into the simulation trace with heat dissipation data in correspondent processor units, taking into account the thermal processes occurring in the processor during the simulation run-time. A heat dissipation data are written to the pre-defined entries in a trace file assigned to individual units of a processor.

Heat trace file contains information (in text form) on heat dissipation in various processor blocks.

After completion of SPEC benchmark run on M-Sim simulator, the trace files are processed and visualized offline by PowerVisor. Figure 2 shows an example of a shared processor snapshot, as depicted by PowerVisor. Here, different colors represent different levels of energy consumption.

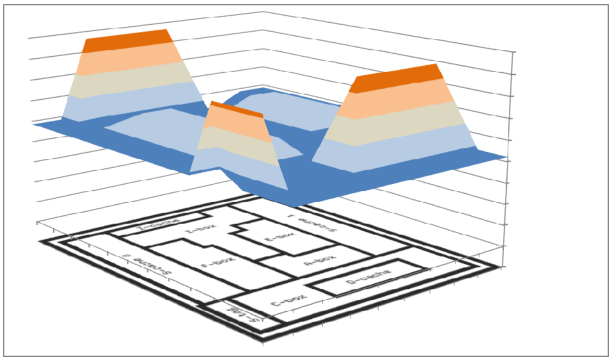


Fig. 2. Energy consumption visualization in PowerVisor tool

While the implementation of the power statistics collection module depends significantly on a particular simulator, PowerVisor 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. Trace files are stored using XLS format which is particularly useful for visual representation and for finding the necessary data. This approach allows for easy integration of PowerVisor tool with processor simulators other than M-Sim.

4 Example Study with PowerVisor

As an example of PowerVisor use in a processor architecture study, we present diagrams of power consumption in a single-core processor obtained from benchmark combination run on M-Sim processor simulator. In this particular case (fig. 3), a series of 3D surfaces (charts) represents changes of power consumption pattern in a processor over time while running SPEC 2006 benchmark combination of GCC and BZIP applications on a cycle-accurate M-Sim processor simulator.

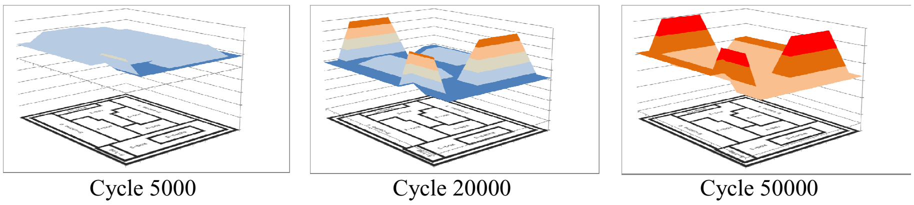


Fig. 3. Energy consumption for GCC and BZIP benchmarks on a single core processor

From these examples one can easily observe the level of non-uniform heating in different parts of the processor, indicating the non-uniform use of those parts during application runs. It can be demonstrated that different combinations of benchmarks (or loads) lead to different patterns of power consumption and heat distribution among the various blocks of the processor, which can provide some clues to processor designers for better layout designs of the CPUs. These data can also be used in efforts on improvement of power and cooling units of new processors. On the other (software) side of the problem, the visual data of power consumption and heat dissipation can be useful for the code optimization efforts to provide a more balanced use of various resources in processors.

5 Concluding Remarks

PowerVisor is an example of data visualizer for study computer architectures. In particular, students and researchers can use PowerVisor (or similar tools) to better understand energy consumption and heat dissipation processes and patterns in the processor blocks or blocks around the processor in the computer system to make changes to the heat dynamics, or to improve the system power consumption. Naturally, PowerVisor can and should be used for educating students in computer architecture classes to better illustrate the content of the course work especially related to power systems and heat dissipation in processors. In fact, in our future plans the development of a web interface to PowerVisor visualization tools is considered to allow remote PowerVisor call directly from your web browser. Our immediate future work involves increasing PowerVisor capacity in modeling and visualization including its integration with Cache Visor toolset [8].

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., et al.: Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. IEEE Micro (November/December 2000)
4. Brooks, D., Martonosi, M.: Dynamic Thermal Management for High-Performance Microprocessors. In: 7th Int'l Symp.High-Performance Computer Architecture, HPCA (2001)
5. 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
6. Sinharoy, B.: Power 7 Multicore Processor Design. Keynote given at the International Symposium on Microarchitecture (2009)
7. M-sim Simulator. Source code and documentation, http://www.cs.binghamton.edu/~msim
8. Evtyushkin, D., Panfilov, P., Ponomarev, D.: CacheVisor: A toolset for visualizing shared caches in multicore and multithreaded processors. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 284–289. Springer, Heidelberg (2011)

SVM Regression Parameters Optimization Using Parallel Global Search Algorithm

Konstantin Barkalov, Alexey Polovinkin, Iosif Meyerov, Sergey Sidorov, and Nikolai Zolotykh

N.I. Lobachevsky State University of Nizhni Novgorod, Nizhni Novgorod, Russia
barkalov@fup.unn.ru, alexey.polovinkin@itlab.unn.ru, {meerov,zolotykh}@vmk.unn.ru, sidorov.sergey@gmail.com

Abstract. The problem of optimal parameters selection for the regression construction method using Support Vector Machine is stated. Cross validation error function is taken as the criterion. Arising bound constrained nonlinear optimization problem is solved using parallel global search algorithm by R. Strongin with a number of modifications. Efficiency of the proposed approach is demonstrated using model problems. A possibility of the algorithm usage on large-scale cluster systems is evaluated. Linear speed-up of combined parallel global search algorithm is demonstrated.

Keywords: Machine learning, global optimization, support vector machine, global search algorithm.

1 Introduction

The problem of reconstructing a functional dependency of a given value on measurement results (regression reconstruction problem) is often met in applied studies. In practice the nature of the given dependence is usually unknown, therefore the unknown function is chosen from a certain pre-determined class which depends on the selected model. Parameters of this function are calculated based on the available experimental data. Among such models [1] we could note polynomial regression, multivariate adaptive regression splines (MARS), radial basis function, decision trees and their ensembles, various boosting algorithms, neural network, etc. An experimental comparison of a number of models and algorithms is given, for instance, in [8].

One of the widely used solution methods for the regression reconstruction problem is the Support Vector Machine (SVM) algorithm [14]. The possibility of efficient nonlinear dependencies modeling and independence of generalization capability from the feature space dimension could be marked out among this algorithm's advantages. Though, in some cases practical application of the algorithm is limited due to the fact that accuracy of the method strongly depends on its parameters selection [3]. Most frequently used approaches to optimal parameter selection are eventually reduced to global optimization problem solution. Thus, for instance, [3] offers to use a genetic algorithm and particle swarm optimization, [9] uses chaos optimization algorithm [10], [11] describes application of a modification of the Efficient Global Optimization (EGO) algorithm [12], [13] uses Pattern Search approach, etc.

As is known, complexity of a global optimization problem solution is significantly influenced by its dimension. For example, for the class of multi-extremal functions satisfying Lipschitz condition the so called curse of dimensionality takes place which represents exponential growth of computing time as a function of dimensionality. Thus, if p calculations of function value are required to reach ε accuracy of solution in a one-dimensional problem, αp^N trials are required to reach the same accuracy in an N-dimensional problem, where α depends on the objective function and on the optimization algorithm in use. Virtually the only way of solving problems of the mentioned class in a reasonable time is the development of parallel algorithms and their implementation on high performance computing systems.

This paper introduces a novel method of SVM regression parameters selection based on optimization of the cross validation error function using parallel global search algorithm. This algorithm is based on the informational statistical approach [5] and as experimentally proven in [5], [7] outperforms many known methods of similar purpose. The paper is organized as follows: an SVM regression parameters optimization problem is stated in the second section; section three describes the basic parallel global search algorithm [5] and its modifications which increase parallel computing efficiency; in section four successfulness of the described approach is demonstrated based on model problems, the possibility of the approach usage on large-scale cluster systems is discussed.

2 Optimization of SVM Regression Algorithm Parameters

This paper considers the regression reconstruction problem in the following statement. Let a training set $D = \{(x_i, y_i), i = 1,\dots,N\}$ be given, where $x_i \in R^d$ is the feature vector, $y_i \in R$ – the response. It is required to find a function $f(x)$ from some specific class K which minimizes the value of empirical risk (prediction error on the training set). For the SVM-regression construction algorithm the function $f(x)$ can be written in general form as

$$f(x) = w^T \phi(x) + b\,,$$

where $\phi(x)$ is a nonlinear (in general case) mapping $R^d \to R^m$, $w \in R^m$ – a vector of linear function coefficients in the new feature space R^m. As a loss function a piecewise linear function of ε-sensitivity is used

$$L_\varepsilon(y, f(x)) = \max(0, |y - f(x)| - \varepsilon)\,,$$

where ε – a predetermined threshold (if the predicted value differs from the actual value less than given threshold the error is considered equal zero). The function of empirical risk is written as:

$$R_{emp}(w) = \frac{1}{N}\sum_{i=1}^{N} L_\varepsilon(y_i, f(x_i))\,.$$

Considered problem of empirical risk minimization is reduced to a quadratic optimization problem

$$\min_{w} \frac{\|w\|^2}{2} + C\sum_{i=1}^{N}\left(\xi_i + \xi_i^*\right)$$
$$f(x_i) - y_i \le \varepsilon + \xi_i,\ y_i - f(x_i) \le \varepsilon + \xi_i^*,\ \xi_i, \xi_i^* \ge 0 \tag{1}$$

where C is a regularization parameter which represents the trade-off between the model complexity and the empirical error in the minimized function. Using the Lagrange multiplier method the problem (1) can be reduced to a dual form:

$$\max \sum_{i=1}^{N} y_i\left(\alpha_i^* - \alpha_i\right) - \varepsilon\sum_{i=1}^{N}\left(\alpha_i^* + \alpha_i\right) - \frac{1}{2}\sum_{i,j=1}^{N}\left(\alpha_i^* - \alpha_i\right)\left(\alpha_j^* - \alpha_j\right)K(x_i, x_j)\ ,$$
$$\sum_{i=1}^{N}\left(\alpha_i^* - \alpha_i\right) = 0,\ 0 \le \alpha_i, \alpha_i^* \le C \tag{2}$$

where α_i, α_i^* – Lagrange multipliers, $K(x_i, x_j)$ – kernel function representing inner product in the new feature space R^m. Some of the most often used kernels are radial basis functions:

$$K\left(x_i, x_j\right) = \exp\left(-\left\|x_i - x_j\right\|^2 / 2\sigma^2\right).$$

As shown in [3] generalization capability of SVM-regression algorithm significantly depends on the choice of parameters C, ε and σ. In paper [2] a method based on cross-validation error minimization is offered. The idea of the method is to split the training set randomly into S subsets $\{G_s, s = 1,...,S\}$, train the model on $(S-1)$ subsets and use the remaining subset to calculate the test error. The error averaged over all training subsets is used as an estimate of algorithm's generalization capability

$$MSE_{CV} = \frac{1}{N}\sum_{s=1}^{S}\sum_{i \in G_s}\left(y_i - f(x_i \mid \theta_s)\right)^2 ,$$

where θ_s is the solution of the problem (2) achieved using the set $D \backslash G_s$ as a training set. In case the number of objects in the training set is not too large LOO (leave-one-out) error can be used

$$MSE_{LOO} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - f(x_i \mid \theta_i)\right)^2 ,$$

where θ_i is the solution of the problem (2) achieved using the set $D \backslash \{(x_i, y_i)\}$ as a training set. Due to the fact that a solution of a quadratic programming problem (2) for each parameter set (C, ε, σ) exists and is unique we can consider the leave-one-out error for a given training set as a function of C, ε and σ:

$$MSE_{LOO} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - f(x_i \mid \theta_i(C, \varepsilon, \sigma))\right)^2 = F(C, \varepsilon, \sigma),$$

Thus the problem of optimal parameter selection for the SVM-regression construction algorithm has been reduced to a problem of function $F(C,\varepsilon,\sigma)$ minimization. In a general case the function is multi-extremal so global optimization algorithms should be applied to find its optimum.

Let us note that the usage of cross-validation method for finding optimal parameter values is traditional in machine learning [15]. Nevertheless, with a large number of determined parameters it can lead to overfitting [16]. In our case only 3 parameters are used which allows us to hope that overfitting won't happen.

3 Parallel Global Search Algorithm

We will find the optimal value of the parameters C, ε and σ in the hypercube $D=[C_{\min};C_{\max}]\times[\varepsilon_{\min};\varepsilon_{\max}]\times[\sigma_{\min};\sigma_{\max}]$. Let us define $\varphi(y)=F(C,\varepsilon,\sigma)$, where $y=(C,\varepsilon,\sigma)$.

Without loss of generality we can consider an unconstrained global optimization problem having the form

$$\begin{gathered}\varphi^*=\varphi(y^*)=\min\{\varphi(y):y\in D\}\\ D=\{y\in R^N:-2^{-1}\le y_i\le 2^{-1},1\le i\le N\}\end{gathered} \tag{3}$$

where the objective function $\varphi(y)$ satisfies the Lipschitz condition

$$|\varphi(y_1)-\varphi(y_2)|\le K\|y_1-y_2\|,\ \ y_1,y_2\in D$$

with constant K which in a general case in unknown. This statement covers a wide class of problems as any hyper interval S

$$S=\{y\in R^N:a_i\le y_i\le b_i,1\le i\le N\}$$

can be reduced to a hyper cube D using a linear coordinate transformation.

In the discussed approach [5] solution of multidimensional problems is reduced to solution of equivalent one-dimensional problems (*dimension reduction*). Thus, the usage of a continuous single-valued mapping such as Peano curve

$$\{y\in R^N:-2^{-1}\le y_i\le 2^{-1},1\le i\le N\}=\{y(x):0\le x\le 1\}$$

allows to reduce the minimization problem in domain D to a minimization problem on the segment [0,1].

$$\varphi^*=\varphi(y^*)=\varphi(y(x^*))=\min\{\varphi(y(x)):x\in[0,1]\}.$$

Numerical methods which allow efficient construction of such mappings with any given accuracy are considered in details in [5]. According to those methods this dimension reduction scheme maps a multidimensional problem with Lipschitz minimized function to one-dimensional problem in which the function satisfies the uniform Hölder condition

$$|\varphi(y(x_1)) - \varphi(y(x_2))| \leq G|x_1 - x_2|^{1/N}, \; x_1, x_2 \in [0,1]$$

where N is the dimension of the original multidimensional problem and Holder coefficient G is connected with Lipschitz constant K of the original problem by the relation $G \leq 4K\sqrt{N}$.

To solve the arising one-dimensional problem it is suggested to use an efficient information-statistical global search algorithm [5]. But when solving the reduced problem with a single scanning part of proximity information for the points in multidimensional space is lost. It is explained by the fact that the point $x \in [0,1]$ has only left and right neighbors while the corresponding point $y(x) \in R^N$ has neighbors along $2N$ directions. Part of points' proximity information can be preserved by using a set of mappings

$$Y_L(x) = \{y^1(x), ..., y^L(x)\} \tag{4}$$

Instead of using a single Peano curve $y(x)$. Each Peano curve $y^i(x)$ from $Y_L(x)$ can be achieved as a result of some transformation of the original curve (shift along the main diagonal of the hypercube [4] or rotation about the origin of the coordinates [7]). The set of Peano curves constructed in this fashion allows to achieve close inverse images x', x'' for any close images y', y'' differing along only one coordinate for some mapping $y^i(x)$.

Usage of the mappings set (4) leads to forming a corresponding set of one-dimensional problems

$$\min\{\varphi(y^l(x)) : x \in [0,1]\}, \; 1 \leq l \leq L. \tag{5}$$

Each problem from this set can be solved independently on a separate processor using the scanning $y^s, 1 \leq s \leq L$. The result of the minimized function $\varphi(y(x))$ value calculation at point x^k received by a particular processor for its solved problem is interpreted as the results of calculations for all remaining problems (in corresponding points $x^{k1}, ..., x^{kL}$) and is sent to other processors. Within such approach a *trial* at point $x^k \in [0,1]$ conducted in problem l consists of the following steps:

1. Determine the image $y^k = y^l(x^k)$ with scanning $y^l(x)$;
2. Inform the rest of the processors of the trial start at point y^k (*blocking* of point y^k);
3. Calculate the value $\varphi(y^k)$. The pair $\{y^l(x^k), z^k = \varphi(y^l(x^k))\}$ is *the result of the trial* at point x^k;

4. Determine inverse images $x^{kl} \in [0,1], 1 \le l \le L$ of the point y^k.
5. Interpret the trial at point $y^k \in D$ as trials at L points $x^{k1},...,x^{kL}$ with same results $\varphi(y^1(x^{k1}) = ... = \varphi(y^L(x^{kL})) = z^k$, i.e. inform the rest of the processors of the trial results at point y^k having sent them pairs (y^k, z^k).

Such informational unity allows to solve the original problem (3) by parallel solving of L problems of form (5) on a set of segments [0,1]. Each processor has an own copy of the software which implements calculation of problem's functions and the decision rule of the algorithm. To organize the communication a queue is created on each processor where processors store information about the performed iterations as pairs: the point of an iteration and minimized function value at that point. Computing scheme of the algorithm is given below.

Algorithm. Starting iteration is performed at an arbitrary point $x^1 \in (0,1)$ (starting points are different for all processors). The choice of the point x^{q+1}, $q \ge 1$, for any subsequent trial on the processor l is defined by the following rules.

Rule 1. For the queue assigned to a given processor remove stored for the processor results including the set Y_q of iteration points in domain D and function values computed for those points. Determine the set X_q of inverse images for the points of set Y_q with scanning $y^l(x)$. Capacity of sets Y_q and X_q is the value s_q such that $s_q \ge q$ as these sets contain the points computed on this processor and received from other processors.

Rule 2. Enumerate the points of iterations set $\{x^1\} \cup X_q$ using subscripts in order of increasing coordinate values

$$0 = x_0 < x_1 < ... < x_i < ...x_k < x_{k+1} = 1, \tag{6}$$

where $k = s_q + 1$, and match them with values $z_i = \varphi(y^l(x_i))$ calculated in these points and integer values $\nu(x_i)$ – the *index* of a point. The index of a non-blocked point x_i (i.e. the point for which results of the trial have been already received) is taken to be equal to 1 while the index of a blocked point x_i (i.e. the point for which the trial has been started by another processor) is taken to be equal to 0, the value z_i is undefined in this case. The points x_0, x_{k+1} are additionally introduced (they do not participate in the trial), indices of these points are taken to be equal to 0 and the values z_0, z_{k+1} are taken as undefined.

Rule 3. Calculate the current lower bound

$$\mu = \max\left\{ \frac{| z_i - z_{i-1} |}{(x_i - x_{i-1})^{1/N}} : 2 \le i \le k, \nu(x_i) = \nu(x_{i-1}) = 1 \right\} \tag{7}$$

for relative differences of function φ. If the value μ turns out undefined (due to unsatisfiability of indices equality conditions from (7)), or if $\mu = 0$ take $\mu = 1$.

Rule 4. For each interval (x_{i-1}, x_i), $1 \le i \le k+1$ calculate the characteristic $R(i)$, where

$$R(i) = \Delta_i + \frac{(z_i - z_{i-1})^2}{r^2 \mu^2 \Delta_i} - 2\frac{(z_i + z_{i-1})}{r\,\mu},\ \nu(x_{i-1}) = \nu(x_i) = 1,$$

$$R(i) = 2\Delta_i - 4\frac{z_i}{r\,\mu},\quad \nu(x_{i-1}) < \nu(x_i),$$

$$R(i) = 2\Delta_i - 4\frac{z_{i-1}}{r\,\mu},\quad \nu(x_{i-1}) > \nu(x_i),$$

$$\Delta_i = (x_i - x_{i-1})^{1/N}$$

The value $r > 1$ is a parameter of the algorithm.

Rule 5. Determine the interval (x_{t-1}, x_t) which has the maximum characteristic

$$R(t) = \max\{R(i) : 1 \le i \le k+1\} \tag{8}$$

Rule 6. Conduct the next trial at the middle point of the interval (x_{t-1}, x_t) if indices of its end points are not equal, i.e.

$$x^{q+1} = \frac{x_t + x_{t-1}}{2},\ \nu(x_{t-1}) \ne \nu(x_t).$$

Otherwise conduct a trial at point

$$x^{q+1} = \frac{x_t + x_{t-1}}{2} - \mathrm{sign}(z_t - z_{t-1})\frac{1}{2r}\left[\frac{|z_t - z_{t-1}|}{\mu}\right]^N,$$

$$\nu(x_{t-1}) = \nu(x_t).$$

Store the results of the trial in the queue assigned to the given processor. Increment q by 1 and move to the next iteration.

Described rules can be augmented with a stopping condition which stops trials if $\Delta_t \le \varepsilon$ where t is from (8) and $\varepsilon > 0$ has the order of the desired per-coordinate accuracy in problem (3)

The following convergence condition is satisfied for this algorithm (as a special case of more general theorem about convergence of a parallel global search algorithm using a set of scannings from [5]).

Convergence Conditions. Let y^* be the point of absolute minimum of a Lipschitz with a constant *K* function $\varphi(y)$, $y \in D$, and $\{y^k\}$ is a sequence of trials generated by a parallel global search algorithm during this function minimization. Then

1. If y', y'' are two limit points of the sequence $\{y^k\}$, then $\varphi(y') = \varphi(y'')$.
2. If y' is a limit point of the sequence $\{y^k\}$, then $\varphi(y') \le \varphi(y^k)$.
3. If at some $k > 0$ the following condition is satisfied for the value μ from (7)

$$r\mu > 8K\sqrt{N},$$

where $r > 1$ is a parameter of the algorithm, then y^* is the limit point of the sequence $\{y^k\}$, and $\lim\limits_{k\to\infty} y^k = y^*$ if y^* is the only point of absolute minimum.

More general variants of parallel global search algorithms (for solution of conditional and multicriteria problems) and corresponding convergence theory are presented in [5-7].

Perspectives of Usage in Parallel Computing. The analysis shows that the described basic algorithm has limitations on the number of used computing devices while using shift scannings [4] (shift along the main diagonal of the hyper cube) as well as while using rotated scannings [7] (rotation about the origin of the coordinates).

In the first case the set of scannings comes out as a result of a shift along the main diagonal of the hyper cube *D* and the step of this shift decreases by 2 times at the construction of each next mapping. The scanning itself is constructed with the accuracy $\delta=2^{-m}$ along the coordinate; *m* here is a parameter of the scanning construction. Construction of a very accurate scanning will require significant resources and the value δ will eventually be limited by the machine error. To solve the problem with accuracy from 10^{-3} to 10^{-4} it is sufficient to choose *m* in the range from 10 to 15 (if necessary, further refinement of the found solution can be performed using one of the known local methods).

As follows from the above the number of scanning shifts during construction of mappings set (4) will be limited by the accuracy of the original scanning construction. If the shift is made by the value smaller than $\delta=2^{-m}$ the next scanning will match the previous one. This way the limitation $L \le m$ is put on the number of used scannings *L* and, as a consequence, on the number of problems solved in parallel.

In the second case the number of processors is limited by the number of possible rotations of an scanning about the origin of the coordinates. In total there can be up to 2^N of such rotations (where *N* is the dimension of the solved problem). But to apply them all would be redundant, rotations in each of coordinate planes are sufficient and these make up $N(N-1)+1$ mappings. This relation gives potential for parallelism at large *N* and limits parallelization possibilities of in problems of smaller dimensions.

A promising in terms of parallelization computing scheme can be based on the approach described in [17]. The essence of the approach consists in conducting simultaneous p trials on intervals which have p greatest characteristics rather than conducting a single trial on interval having the maximum characteristic (8). This approach can be applied together with the described above mappings method: conduct p trials in parallel for each problems from the set (5) solution of which is also performed in parallel [18].

Mentioned combined algorithm can be used on modern cluster systems with multi-core processors. An own sub-problem (5) is assigned to each node and the number of simultaneous trials for one problem corresponds to the number of cores on the node.

4 Computational Experiment

4.1 Optimization of a Function of Two Variables: Comparison with the Exhaustive Search

In this section we will demonstrate the efficiency of the basic algorithm for SMV-regression parameters optimization on a two-dimensional problem example. We will also compare the algorithm with the exhaustive search. Consider the following problem [2]. Let the function $y(x) = 0.5 + 0.4\sin(2\pi x)\cos(6\pi x)$ be given. $x_i = 0.05\cdot(i-1), i = 1,...,21$, $y_i = y(x_i) + N(0,0.05)$, where $N(0,0.05)$ – a Gaussian distribution with the mean equal to 0 and standard deviation equal to 0.05. Figure 1 presents the plot of the cross-validation error MSE_{LOO} dependency on ε and σ with fixed value of parameter $C = 1$. As seen from the plot the function describing this dependency has several local minima in the search domain.

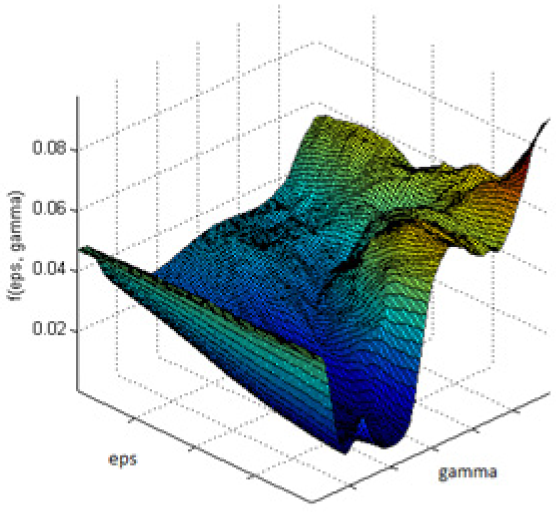


Fig. 1. Dependency of cross-validation error on SVM-regression parameters

The algorithm was run with the following parameters: the accuracy of the search $\rho = 0.01$, reliability parameter $r = 2.5$, the accuracy of the scanning construction along each coordinate was $2.5 \cdot 10^{-4}$, the number of scannings $L = 2$. The total number of iterations performed by the parallel version of the algorithm before reaching stopping criteria was 1550 (to reach the same accuracy with the exhaustive search 10000 iterations would be required), found optimum value was 0.006728. The experiment showed significant advantage of the algorithm over the exhaustive search. With increasing dimension of the problem the gap in the number of performed iterations and thus the gap in solution time will only be growing.

4.2 Optimization of a Function of Two Variables: Efficiency of the Algorithm and Its Modifications When Using Parallel Computing

In this section we consider a possibility of using different modifications of parallel global search algorithm when optimizing parameters of SVN-regression. We will use the following described above implementations of the algorithm:

1. Basic parallel global search algorithm with shift scannings.
2. Parallel global search algorithm with rotated scannings.
3. Combined parallel global search algorithm.

Taking into account considerations on usage perspectives of implementations 1-3 in parallel computing let us ascertain the applicability of the most promising implementation 3, also let us compare the problem solution times using same number of scannings *L* in all implementations. In addition accounting for the fact that the number of scannings in the second implementation is limited by the value $N(N-1)+1$ where *N* is the problem dimension we will take $L = 3$. Having this setting implementations 1 and 2 will allow to use 3 cores each and implementation 3 will allow to use $3p$ cores (3 rotated scannings and p cores for each of the scannings).

Consider the following problem: let a real function of n real variables be given having the form: $y(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} w_i \sin(\alpha_i \pi x_i)\cos(\beta_i \pi x_i)$, where parameters $w_i \sim Uniform(0,1)$, $\alpha_i \sim Uniform(2,8)$, $\beta_i \sim Uniform(2,8)$, $i = 1,...,n$, $Uniform(a,b)$ – a uniform distribution on segment $[a;b]$. Consider the case $n = 3$, $y_{ijk} = y(x_1^i, x_2^j, x_3^k) + N(0,0.3)$, where $x_1^i = 0.1 \cdot (i-1)$, $i = 1,...,11$, $x_2^j = 0.1 \cdot (j-1)$, $j = 1,...,11$, $x_3^k = 0.1 \cdot (k-1)$, $k = 1,...,11$. This way the training set $(y_{ijk}, x_1^i, x_2^j, x_3^k)$ contains $11^3 = 1331$ objects.

Consider an optimization problem of the cross-validation error MSE_{LOO} of ε and σ with fixed value of parameter $C = 5$.

The cluster of the University of Nizhni Novgorod was used to conduct experiments. A node of the cluster contains 2x Intel Xeon 5150 (total 8 cores), 24GB RAM, Microsoft Windows HPC Server 2008.

Let us study the results of the experiments. In all runs the following parameters were used: search accuracy $\rho = 0.01$, reliability parameter $r = 2$. Let us note that solving the given problem with the specified accuracy using full search over a uniform grid would require 10^4 iterations which is significantly more than the number of iterations spent on optimum search using the considered global search algorithm. All its implementations 1-3 have reached the required accuracy after 400 iterations, obtained values of the optimum were 0.090543, 0.090564 and 0.09054. Working times of the algorithms are given in the table below.

Table 1. Running time of the algorithms (optimization of a function of two variables)

Implementation	Scannings	Cores on scanning	Time (hours)
1	3	1	2,4
2	3	1	2,4
3	3	2	1,2
3	3	4	0,6

Implementation 3 showed 2x performance advantage compared to the other implementations having used 2 times more cores. Taking into account that implementations 1 and 2 are substantially limited in ability of parallel usage of computing devices, implementation 3 (having same computing time per core) allows usage of bigger number of cluster nodes which leads to computing time decrease.

4.3 Optimization of a Function of Three Variables: Efficiency of the Algorithm and Its Modifications When Using Parallel Computing

This section presents scalability of implementation 3 measured on the cluster of the University of Nizhni Novgorod. We solved the optimization problem on three parameters stated in the previous section, the value of parameter C is no longer fixed. Accuracy of the optimum search (along the coordinate) in this series of experiments was $\rho = 0.02$. It was sufficient to make 1000 iterations to achieve the desired search accuracy. Taking into account that the number of scannings is limited by $N(N - 1) + 1$, where N=3 (the problem dimension), we ran the algorithms to test scalability in the following configurations: using L=3 scannings on three nodes and using L=6 scannings on six cluster nodes (implementation 2 of the algorithm), using L=6 scannings on six cluster nodes and 2 or 4 cores for each of the scannings (implementation 3 of the algorithm). All algorithms found the optimal value 0.09054 of the objective function, working time of the algorithms is given in the table below.

Table 2. Running time of the algorithms (optimization of a function of three variables)

Implementation	Scannings	Cores on scanning	Time (hours)
2	3	1	5.5
2	6	1	2.8
3	6	2	1.4
3	6	4	0.8

The results demonstrate a linear speedup in the number of used nodes and cores. More cluster nodes could be used in case of a larger number of SVM parameters (which depends on used kernel).

5 Conclusions

This paper considers the problem of optimal parameter selection for the regression construction method using support vector machines. Cross-validation error function was chosen as the optimized criteria in the given problem. Arising problem belongs to the class of bound constrained nonlinear optimization, objective function is often multi-extremal which stipulates the necessity of global optimization methods application.

For the solution of the problem the paper suggests to use global search methods presented in studies of R. Strongin, V. Gergel, V. Grishagin, Ya. Sergeev, et al., with a number of modifications (rotated scannings, combined scheme of parallel computing). The questions of parallel computing usage are considered within this approach. The combined scheme was proved to be the most suitable for cluster systems with large number of multicore processors due to the possibility of all computing resources utilization. Computational experiments were conducted to support the stated propositions. Three schemes of parallel implementation of the global search algorithm are analyzed. Scalability of the first two schemes is limited by the number of scannings used by the algorithm. The combined scheme of the parallel global search algorithm demonstrates linear speedup in SVM-regression parameters optimization, lets use multicore processors effectively and therefore scales much better.

Acknowledgements. This study was conducted with the support of the Ministry of Education and Science of Russian Federation, the program "Research and development in priority directions of Russia science and technology complex", contract 11.519.11.4015; the program "Scientific and academic human resources of innovation Russia", contract 14.B37.21.0393.

The authors thank Alexander Sysoyev, Alexey Sidnev, Vasily Ryabov and Nina Kurina for help and useful discussions.

References

1. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference and Prediction. Springer, Heidelberg (2009)
2. Ito, K., Nakano, R.: Optimization Support Vector Regression Hyperparameters Based on Cross-Validation. In: Proceedings of the International Joint Conference on Neural Networks, vol. 3, pp. 2077–2083 (2003)
3. Ren, Y., Bai, G.: Determination of Optimal SVM Parameters by Using GA/PSO. Journal of Computers 5(8), 1160–1168 (2010)
4. Strongin, R.G.: Algorithms for multi-extremal mathematical programming problems employing the set of joint space-filling curves. J. of Global Optimization 2, 357–378 (1992)

5. Strongin, R.G., Sergeyev, Y.D.: Global optimization with non-convex constraints. Sequential and parallel algorithms. Kluwer Academic Publishers, Dordrecht (2000)
6. Gergel, V.P., Strongin, R.G.: Parallel computing for globally optimal decision making on cluster systems. Future Generation Computer Systems 21(5), 673–678 (2000)
7. Barkalov, K., Ryabov, V., Sidorov, S.: Parallel Scalable Algorithms with Mixed Local-Global Strategy for Global Optimization Problems. In: Hsu, C.-H., Malyshkin, V. (eds.) MTPP 2010. LNCS, vol. 6083, pp. 232–240. Springer, Heidelberg (2010)
8. Jin, R., Chen, W., Simpson, T.W.: Comparative Studies of Meta-modeling Techniques under Multiple Modeling Criteria. Struct. Multidiscip. Optim. 23(1), 1–13 (2001)
9. Wang, Y., Liu, Y., Ye, N., Yao, G.: The Parameters Selection for SVM Based on Improved Chaos Optimization Algorithm. In: Zhang, J. (ed.) ICAIC 2011, Part V. CCIS, vol. 228, pp. 376–383. Springer, Heidelberg (2011)
10. Zhang, H., He, Y.: Comparative study of chaotic neural networks with different models of chaotic noise. In: Wang, L., Chen, K., S. Ong, Y. (eds.) ICNC 2005. LNCS, vol. 3610, pp. 273–282. Springer, Heidelberg (2005)
11. Fröhlich, H., Zell, A.: Efficient parameter selection for support vector machines in classification and regression via model-based global optimization. In: IEEE International Joint Conference on Neural Networks, IJCNN 2005, vol. 3, pp. 1431–1436 (2005)
12. Jones, D., Schonlau, M., Welch, W.: Efficient global optimization of expensive black-box functions. J. Global Optimization 13, 455–492 (1998)
13. Momma, M., Bennett, K.P.: A pattern search method for model selection of support vector regression. In: Proceedings of SIAM Conference on Data Mining, pp. 261–274. SIAM, Philadelphia (2002)
14. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, New York (1996)
15. Kearns, M.: A bound on the error of cross validation using the approximation and estimation rates, with consequences for the training-test split. In: Adv. In Neural Information Processing Systems, vol. 8, pp. 183–189. MIT Press (1996)
16. Ng, A.Y.: Preventing of overfitting of cross-validation data. In: Proc. 14th Int. Conf. on Machine Leaning, pp. 245–253. Morgan Kaufmann (1997)
17. Grishagin, V.A., Sergeyev, Y.D., Strongin, R.G.: Parallel characteristical global optimization algorithms. Journal of Global Optimization 10(2), 185–206 (1997)
18. Sidorov, S.V.: Two-level parallel index algorithm for global optimization. Vestnik of Lobachevsky State University of Nizhni Novogorod 5(2), 208–213 (2012) (in Russian)

Secure and Unfailing Services[⋆]

Davide Basile, Pierpaolo Degano, and Gian-Luigi Ferrari

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

Abstract. Internet is offering a variety of services, that are assembled to accomplish requests made by clients. While serving a request, security of the communications and of the data exchanged among services is crucial. Furthermore, communications occur along specific channels, and it is equally important to guarantee that the interactions between a client and a server never get blocked because either cannot access a selected channel. We address here both these problems, from a formal point of view. A static analysis is presented, guaranteeing that a composition of a client and of possibly nested services respects both security policies for access control, and compliance between clients and servers.

1 Introduction

The opportunities of exploiting distributed services are becoming an imperative for all organizations and, at the same time, new programming techniques are transforming the ways distributed software architectures are designed and implemented. Distributed applications nowadays are constructed by assembling together computational facilities and resources offered by (possibly) untrusted providers. For example in Service-Oriented Computing, the basic building blocks are independent software components called services, equipped with suitable interfaces describing (roughly) the offered computational facilities. Moreover, standard communication protocols (e.g. SOAP over HTTP) take care of the interaction between the parties. Another illustrative example of this approach for effective distributed services exploitation is the so-called Cloud computing.

Software architectures that truly support distributed services require several operational tools to be in place and effective. In these computing environments, managing security issues is rather complex, since security controls are needed to control access to services and resources, as well as to create suitable services on demand. For example, identity management and authorization are vital to create and deploy services on-the-fly. Moreover, when services are made available through third parties, understanding and fulfilling the behavioural obligations of services is crucial to determine whether the interactive behaviour is consistent with the requirements. It is of course important to ensure that clients and services interact correctly, which in turn implies verifying the correctness of their behavioural obligations. *Service Contracts* are the standard mechanisms to describe the external observable behaviour of a service, as well as the responsibility

[⋆] This work has been partially supported by the MIUR project *Security Horizons.*

for security. Service contracts can be used for guaranteeing that all the services are capable of successfully terminating their tasks (progress property) without rising security exceptions.

The management of service contracts is typically tackled by service providers through the notion of *Service-Level Agreement* (SLA), which have to be accepted by the client before using the service. SLAs are documents that specify the level(s) of service being sold in plain language terms. Recently, so called *contract-oriented design* has been introduced as a suitable methodology where the interaction between clients and services is specified and regulated by formal entities, named contracts. The formal management of contracts permits to unambiguously represent the obligations and also supports their automatic analysis and verification. In the literature several proposals have addressed and investigated contracts, exploiting a variety of techniques [9,11,10,13,12,8,1]. The amalgamation of security issues within contract based design has been recently tackled by [6], that develops a calculus where behavioural contracts may be violated by dishonest participants after they have been agreed upon.

In this paper we develop a formal theory of contracts that supports verification of *service compliance*. Services are compliant when their interactive behaviour eventually progresses, i.e. all the service invocations are guaranteed to be eventually served. Moreover, our theory of contracts supports verification of security policies enforcing access control over resources.

Our starting point is the language-based methodology supporting static analysis of security policies developed in [5,4]. The main ingredients of this approach are: local policies, call-by-contract invocation, type and effect systems, model checking and secure orchestration. Then, security policies, expressed as safety policies, are model checked. A plan orchestrates the execution of a service-based application, by associating the sequence of run-time service requests with a corresponding sequence of selected services. A main result shows how to construct a plan that guarantees that no executions will abort because of some action attempting to violate security. In [14] the methodology has been extended to deal with quantitative aspects of service contracts, typically the rates at which the different activities are performed.

Here we extend this approach to deal with service compliance. Our first contribution is to extend the abstraction of service behaviour, called history expressions, with suitable communication facilities to model the interactive behaviour of services, including service sessions in a multiparty fashion. In particular, we extend history expressions to include channel communications and internal/external choice for combining the notions of security access and progress of interactions. Our second contribution is sharpening the verification phase. Roughly, we reduce the problem of checking service compliance to that of checking a safety property, expressing that all the involved parties are able to successfully terminate their interactions without getting stuck. Reducing service compliance to a safety property makes it model-checkable. Finally, we extract from a history expression all the viable plans, i.e. those orchestrations that successfully drive secure and compliant executions. Adopting a valid plan guarantees

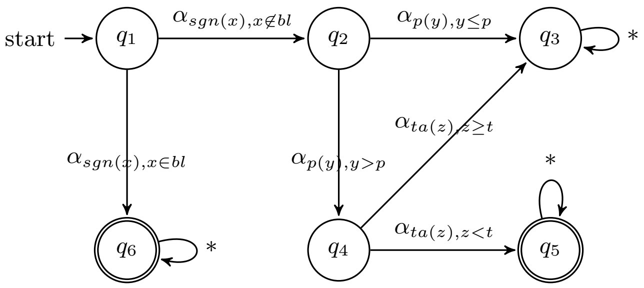


Fig. 1. The automaton for the policy $\varphi_{(bl,p,t)}$

that the involved services never go wrong at run-time, in that they are capable of successfully accomplishing their tasks (progress property) without raising any security exceptions. Therefore, no run-time monitor is needed, to control the execution of the network of services.

The paper is organised as follows. The next section intuitively presents our formalism, the problems and our goal through an example. Sect. 3 formalizes the main aspects of our proposal, defines the syntax and the semantics of history expressions; it recalls the methodology of [5,4] and shows it applicable to our case for verifying the security aspects of services. The definition of compliance, its reduction to a safety property and the technicalities needed to model-check it are in Sect. 4. In Sect. 5, we discuss our results and future work.

2 Motivating Example

To illustrate our approach and help intuition, we propose an example of a simple service for booking hotels. A broker is responsible for collecting the clients' requests and for sending the informations to the hotel companies. When a client opens a session with the broker, he puts some constraints on the quality of service. This are represented by the policy $\varphi_{(bl,p,t)}$. Roughly, policies are regular properties over the execution histories, which are specified through the so-called usage automata [3], a sort of parametric finite state automata. Our policy φ is in Figure 1. Its parameters are a black list of hotels *bl*, a threshold on the price p and on the Trip Advisor's rating t. When selected, an hotel signs the contract, i.e. emits a signal $\alpha_{sgn(x)}$, also called *event*. Then, it publishes its price and rating with the events $\alpha_{p(y)}, \alpha_{ta(z)}$. If the hotel is in the black list the policy is violated and the final state q_6 is reached. Note that the forbidden traces belong then to the language *accepted* by the automaton, as prescribed by the "default-allow" approach. A violation of the policy also occurs if the hotel has a price higher then p and the Trip Advisor's rating is lower then t. As we will see, our approach is history dependent, so all the actions performed so far will be actually inferred.

We specify through a process calculus two clients C_1, C_2, and a repository R including a broker B_1 and four hotels S_1, S_2, S_3, S_4, in Fig. 2. The two clients

$$\begin{aligned}
C_1 &= \mathtt{open}_{1,\varphi(\{s1\},45,100)}\overline{Req}(CoBo.\overline{Pay} + NoAv)\mathtt{close}_{1,\varphi(\{s1\},45,100)}\\
C_2 &= \mathtt{open}_{2,\varphi(\{s1,s3\},40,70)}\overline{Req}(CoBo.\overline{Pay} + NoAv)\mathtt{close}_{2,\varphi(\{s1,s3\},40,70)}\\
Br &= Req.\mathtt{open}_{3,\emptyset}\overline{IdC}.(Bok + UnA)\mathtt{close}_{3,\emptyset}(\overline{CoBo}.Pay \oplus \overline{NoAv})\\
S_1 &= \alpha_{sgn(1)}.\alpha_{p(45)}.\alpha_{ta(80)}.IdC(\overline{Bok} \oplus \overline{UnA})\\
S_2 &= \alpha_{sgn(2)}.\alpha_{p(70)}.\alpha_{ta(100)}.IdC(\overline{Bok} \oplus \overline{UnA} \oplus \overline{Del})\\
S_3 &= \alpha_{sgn(3)}.\alpha_{p(90)}.\alpha_{ta(100)}.IdC(\overline{Bok} \oplus \overline{UnA})\\
S_4 &= \alpha_{sgn(4)}.\alpha_{p(50)}.\alpha_{ta(90)}.IdC(\overline{Bok} \oplus \overline{UnA})
\end{aligned}$$

Fig. 2. Two clients, a broker and four hotels

differ only in the way they instantiate their policies. The client opens a session and sends his request to the broker, who must respect the policy φ. Sending the request is modelled by the action $\overline{Req}$, while receiving the request is done through Req, the complementary action (for the sake of simplicity we omit the data needed for booking the hotel). The client is then willing to receive the confirmation of the booking and to settle the bill ($CoBo.\overline{Pay}$). The client is also ready to receive a negative message of no rooms are available ($NoAv$). When either message is received, the session with the broker is closed. As said, the broker receives the request Req and then opens a session with one of the hotels. The broker sends the Id of the client and all the data with $\overline{IdC}$, and then waits either the booking or the negative message with ($Bok + UnA$). Then the session with the hotel is closed, and the response message is forwarded to the client. The hotel services perform the events of signing and publishing the price and rating; and then interact with the broker. Note that all services, except for S_2, have the *internal* choice $\overline{Bok} \oplus \overline{UnA}$ and decide on their own which message to send. Note that the internal choice is different from the external choice in, e.g. $Bok + UnA$, that is instead driven by the message received. Since Br is ready to receive each sent message, we say that the mentioned services are *compliant* with Br. Instead, service S_2 is *not compliant* with Br since it can send a message $\overline{Del}$ (meaning that there will be available rooms later in the week) that the broker is not able to handle, and the interaction gets stuck. As far as security is concerned, it turns out that the services S_1 and S_4 violate the policy of C_1, since S_1 is black listed and S_4 respects neither thresholds; while the services S_1, S_3 do not satisfy the policy of C_2 since they are black listed.

Figure 3 displays a fragment of a computation. It is a sequence of configurations χ and of transitions $\chi \xrightarrow{\gamma} \chi'$, where γ records either an event or a communication made of two complementary actions (disregard for a while the indexes π, R of the arrows). A configuration is made of tuples $\eta, \ell : S$, put in parallel (through $\|$), where η is a sequence of events, ℓ is the location of the service/client S. In our example, the starting configuration has the two clients, one on location ℓ_{c1}, the other on ℓ_{c2}. Both performed no actions, so their execution history is empty (ε). The first step opens a session between C_1 and Br

$$\varepsilon, \ell_{c1} : C_1 \| \varepsilon, \ell_{c2} : C_2 \xrightarrow{open_{1,\varphi_1}}_{\pi,R}$$

$$(\!|_{\varphi_1}, [\ell_{c1} : \overline{Req}.(CoBo.\ldots)\mathtt{close}_{1,\varphi_1}, \ell_{br} : Br] \| \varepsilon, \ell_{c2} : C_2 \xrightarrow{\tau}_{\pi,R}$$

$$(\!|_{\varphi_1}, [\ell_{c1} : (CoBo.\ldots)\mathtt{close}_{1,\varphi_1}, \ell_{br} : \mathtt{open}_{3,\emptyset}\overline{IdC}\ldots]] \| \varepsilon, \ell_{c2} : C_2 \xrightarrow{open_{3,\emptyset}}_{\pi,R}$$

$$\overbrace{(\!|_{\varphi_1}, [\ell_{c1}\ldots, [\ell_{br} : \overline{IdC}\ldots, \ell_{s3} : \alpha_{sgn(3)}\cdots]]}^{P} \| \varepsilon, \ell_{c2} : C_2 \xrightarrow{open_{2,\varphi_2}}_{\pi,R}$$

$$P \| \overbrace{(\!|_{\varphi_2}, [\ell_{c2} : \overline{Req}\ldots \mathtt{close}_{2,\varphi_2}, \ell_{br} : Br]}^{Q} \xrightarrow{\alpha_{sgn(3)}}_{\pi,R} \xrightarrow{\alpha_{p(90)}}_{\pi,R} \xrightarrow{\alpha_{ta(100)}}_{\pi,R}$$

$$\overbrace{(\!|_{\varphi_1}\alpha_{sgn(3)}\alpha_{p(90)}\alpha_{ta(100)}}^{\eta}, [\ell_{c1} : \ldots, [\ell_{br} : \overline{IdC}\ldots, \ell_{s3} : Idc\ldots]]\|Q \xrightarrow{\tau}_{\pi,R} \xrightarrow{\tau}_{\pi,R}$$

$$\eta, [\ell_{c1} : \ldots, [\ell_{br} : \mathtt{close}_{3,\emptyset}\ldots, \ell_{s3} : \varepsilon]]\|Q \xrightarrow{close_{3,\emptyset}}_{\pi,R}$$

$$\eta, [\ell_{c1} : (CoBo.\overline{Pay} + NoAv)\mathtt{close}_{1,\varphi_1}, \ell_{br} : (\overline{CoBo}.Pay \oplus \overline{NoAv})]\|Q \xrightarrow{\tau}_{\pi,R}$$

$$\eta, [\ell_{c1} : \mathtt{close}_{1,\varphi_1}, \ell_{br} : \varepsilon]\|Q \xrightarrow{close_{1,\varphi_1}}_{\pi,R}$$

$$\eta|\!)_{\varphi_1}, \ell_{c1} : \varepsilon \| (\!|_{\varphi_2}, [\ell_{c2} : \overline{Req}.\ldots, \ell_{br} : Br] \xrightarrow{\tau}_{\pi,R} \ \ldots$$

Fig. 3. A fragment of a computation

and registers in the history that the whole session, in particular Br, is subject to the policy φ, duly instantiated (call it $\varphi_1 = \varphi_{(\{s1\},45,100)}$). The second step shows that the request of the client has been accepted by the broker, via a communication. Now a nested session is opened involving the broker and S_3, in the third step, and no policy is put over the called service S_3.

Concurrently, C_2 can ask for a reservation, as expressed by the fourth step [1], that registers that the policy $\varphi_2 = \varphi_{(\{s1,s3\},40,70)}$ is active. Note that we assume that the broker can replicate its code at will.

The two parallel sessions can evolve concurrently. For simplicity, we proceed with service S_3, that signs, shows the price and its rating (all displayed in the same line). The broker is ready to send the client's data to S_3, and to receive back an answer, say "no room is available" (S_3 is now ε, because it has no further activities to do). The session is then closed in the 10^{th} step, and the broker resumes its conversation with the client C_1, and forwards the non-availability, in step 11. The next steps close the session numbered 1 and the security framing of φ_1. The last transition continues the session involving the second client.

The index R of the arrows shows that the transitions depend on the services contained in the repository R.

The index π is a vector of functions, called a *plan*, that indicates how the requests are bound to services. The plan π_1 for the first client maps the request

[1] Our present framework deals with concurrency in an interleaving style, so the activity of the clients, the broker, the hotels are interleaved; for a more realistic, truly concurrent approach, see [15]).

1, originated by $\texttt{open}_1$ of the client, to ℓ_{br}, and the request 3 from $\texttt{open}_3$ of the broker, to ℓ_{s3}. We call π_1 valid, because it drives a computation where both the security constraints and compliance of clients/services are guaranteed.

Suppose now that the plan π_2 for the second client maps request 2 to ℓ_{br} and request 3 (from the second instance of the broker) to ℓ_{s2}. Since S_2 does not comply with Br, at run-time a communication involving the action $\overline{Del}$ cannot be performed because the broker has no action Del. Our assumption that the service can decide what to send on its own is violated. For this reason, we say that this plan is not valid. Finally, consider a plan that maps request 3 to ℓ_{s3}, that this time is compliant with the broker. However S_3 is black-listed by C_2, and so a policy violation occurs; also this plan is not valid.

Our task in the next sections will be defining a static analysis that allows us to construct valid plans, only. With such plans, neither violations of security, nor missing communications can occur, so there is no need for any execution monitor at run-time.

3 Programming Model

Here we define the syntax and the semantics of services and of networks of services. Services are represented by λ-expressions, and a type and effect system extracts their abstract behaviour, in the form of *history expressions*. Lack of space prevents us from giving a detailed presentation of this model, and only refer the reader to [5,4]. We further extend the basic model of history expressions with standard I/O operations, because we want to explicitly represent through communications also the interactions between clients and services. Moreover, we explicitly deal with sessions, and our history expressions will therefore record also the operations of opening and closing them. Our final extension permits to have several sessions in parallel. In this work, we address neither the analogous extensions to the λ-calculus, nor the definition of a type and effect system for it.

Some auxiliary notions are in order. We assume to have a set of security relevant operations described by events $\alpha \in \mathsf{Ev}$, and a set of policies $\varphi \in \mathsf{Pol}$, i.e. a regular language over Ev. Opening and closing a session is modelled through communication actions, labelled by a request identifier $r \in \mathsf{Req}$ and a policy φ. These special activities will be logged in computations by framing actions $\mathsf{Frm} = \{(\!|_\varphi, |\!)_\varphi | \varphi \in \mathsf{Pol}\}$. We also assume the presence of channels along which clients and services communicate. So we have a set of communication actions $\mathsf{Comm} = \{a, \overline{a}, \tau, \texttt{open}_{r,\varphi}, \texttt{close}_{r,\varphi}\}$, where as usual $\overline{\overline{a}} = a$. Hereafter, let $\mathsf{Act} = \mathsf{Ev} \cup \mathsf{Comm}$ and let $\lambda \in \mathsf{Comm} \cup \mathsf{Ev} \cup \mathsf{Frm}$. Finally, we assume services and clients be hosted in locations $\ell \in \mathsf{Loc}$.

Definition 1 (History Expression). *A history expression is a term generated by the following grammar:*

$$H ::= \varepsilon \mid h \mid \mu h.H \mid (\sum_{i\in I} a_i.H_i) \mid (\bigoplus_{i\in I} \overline{a}_i.H_i) \mid \alpha \mid H \cdot H \mid \mathit{open}_{r,\varphi} H,\ \mathit{close}_{r,\varphi} \mid \varphi(\!|H|\!)$$

Intuitively, ε is the history expression that cannot do anything, and thus we stipulate $\varepsilon \cdot H \equiv H \equiv H \cdot \varepsilon$.

Infinite behaviour is denoted by $\mu h.H$, restricted to be tail-recursive and guarded by communication actions $\bar{a}$ or a. Events α can occur, if they do not violate any active policy. The expression $H_1 \cdot H_2$ is the sequential composition.

An expression can send/receive on a channel messages, indexed by a set I. To stress that the non-deterministic choice of the output $\overline{a}_i$ is up to the sender only (internal choice), we use $\oplus$, while the external choice only involves inputs a_i and is denoted by Σ.

Framing $\varphi(\!|H|\!)$ says that while H is running, the policy φ must be enforced (sometimes we assume $\varphi(\!|H|\!) \equiv (\!|_\varphi \cdot H \cdot |\!)_\varphi$). As anticipated, a policy is a (sort of) finite state automaton that accepts those strings of access events that violate it, in the default-accept paradigm. An example is "never write (α_{write}) after read (α_{read})", and a trace that violates it is $\alpha_{read}\alpha_{write}$. When entering a security framing, all the history, i.e. the sequence of events previously fired, must respect the policy: ours is a *history-dependent* approach. We remark that policies are safety properties: nothing bad occurred so far.

A service is engaged in a session with another through $\texttt{open}_{r,\varphi} H \texttt{close}_{r,\varphi}$, where r is a unique identifier and φ is the policy to be enforced while the responding service is active. Later on we will require that a client must be able to synchronize with the server and correctly terminate the session, i.e. client and service have to be compliant.

The following rules inductively define the semantics of a stand-alone history expression.

Operational Semantics of History Expressions

$$\bigoplus_{i \in I} \overline{a}_i.H_i \xrightarrow{\overline{a}_i} H_i \quad (\text{I-CHOICE}) \qquad \sum_{i \in I} a_i.H_i \xrightarrow{a_i} H_i \quad (\text{E-CHOICE})$$

$$\alpha \xrightarrow{\alpha} \varepsilon \quad (\text{ACC}) \qquad \texttt{open}_{r,\varphi}.H.\texttt{close}_{r,\varphi} \xrightarrow{open_{r,\varphi}} H.\texttt{close}_{r,\varphi} \quad (\text{S-OPEN})$$

$$\varphi(\!|H|\!) \xrightarrow{(\!|_\varphi} H \cdot |\!)_\varphi \quad (\text{P-OPEN}) \qquad \frac{H \xrightarrow{\lambda} H'}{H \cdot H'' \xrightarrow{\lambda} H' \cdot H''} \quad (\text{CONC})$$

$$\frac{H\{^{\mu h.H}/_h\} \xrightarrow{\lambda} H'}{\mu h.H \xrightarrow{\lambda} H'} \quad (\text{REC})$$

We now turn our attention to our specification of networks of services N. In the following definition, we also introduce the notion of plan π.

Definition 2 (Network and Plan)

$$N ::= N \| N \mid S \qquad S ::= \ell : H \mid [S, S]$$

$$\vec{\pi} = [\pi_1, \ldots, \pi_n], \;\; \textit{where } \pi_i, \pi'_i ::= \emptyset \mid r[\ell] \mid \pi \cup \pi'$$

Operational Semantics of Networks

$$\frac{H \overset{open_{r,\varphi}}{\twoheadrightarrow} H' \quad r[\ell_j] \in \pi \quad \{\ell_j : H_j\} \subseteq R \quad \models \eta (\!|_\varphi}{\eta, \ell_i : H \xrightarrow{open_{r,\varphi}}_{\pi,R} \eta (\!|_\varphi, [\ell_i : H', \ell_j : H_j]} \ (\textsc{Open})$$

$$\frac{H \overset{close_{r,\varphi}}{\twoheadrightarrow} H'}{\eta, [\ell_i : H, \ell_j : H''_j] \xrightarrow{close_{r,\varphi}}_{\pi,R} \eta\eta', \ell_i : H'} \ \eta' = \Phi(H''_j) |\!)_\varphi \ (\textsc{Close})$$

$$\frac{\eta, S \xrightarrow{\lambda}_{\pi,R} \eta', S' \quad \models \eta'}{\eta, [S, S''] \xrightarrow{\lambda}_{\pi,R} \eta', [S', S'']} \ (\textsc{Session})$$

$$\frac{\eta_i, N_i \xrightarrow{\lambda}_{\pi_i,R} \eta'_i, N'_i \quad \models \eta'_i \quad (\vec{\pi})_i = \pi_i \quad \overrightarrow{(\eta, N)}_i = \eta_i, N_i}{\overrightarrow{(\eta, N)} \xrightarrow{\lambda}_{\vec{\pi},R} \overrightarrow{(\eta, N)[\eta'_i, N'_i \mapsto \eta_i, N_i]}} \ (\textsc{Net})$$

$$\frac{H \overset{\gamma}{\twoheadrightarrow} H' \quad \models \eta\gamma}{\eta, \ell_i : H \xrightarrow{\gamma}_{\pi,R} \eta\gamma, \ell_i : H'} \ \gamma \in \mathsf{Ev} \cup \mathsf{Frm} \ (\textsc{Access})$$

$$\frac{H_i \overset{a}{\twoheadrightarrow} H'_i \quad H_j \overset{co(a)}{\twoheadrightarrow} H'_J}{\eta, [\ell_i : H_i, \ell_j : H_j] \xrightarrow{\tau}_{\pi,R} \eta, [\ell_i : H'_i, \ell_j : H'_j]} \ a \in \mathsf{Comm} \ (\textsc{Synch})$$

where $\Phi(H_1 \cdot H_2) = \Phi(H_1) \cdot \Phi(H_2) \qquad \Phi(|\!)_\varphi) = |\!)_\varphi \qquad \Phi(H) = \varepsilon$ otherwise

A network N is composed of the parallel composition of different clients H, each hosted at a location $\ell \in \mathsf{Loc}$, and of sessions S involving a client (or a service) and a service. Services are published in a global trusted repository $R = \{\ell_j : H_j \mid j \in J\}$, and they are always available for joining sessions.

We assume that the operator $\|$ is associative, but not commutative, so a network can be written as a vector $\vec{N}$. Instead, we stipulate that $[S, S'] \equiv [S', S]$.

We can have nested sessions, modelling that a service involved in a session can open a new session with another service. In this case the previous session will be restored upon termination of the new one.

The semantic of networks is the transition system, inductively defined by the rules in the table below. Its configurations have the form $\|_{i \in I}\, \eta_i, S_i$ abbreviated by $\overrightarrow{\eta, N}$, where η_i is the *history* of S_i. As a matter of fact, access events α and policy framings $(\!|_\varphi, |\!)_\varphi$ are logged into the history η_i. A session can evolve only if its history respects all the active policies in η_i, denoted by $\models \eta_i$ (see below).

We briefly comment on the rules of the operational semantic of networks. The first rule is for opening a session: the service at ℓ_i fires an event $\mathtt{open}_{r,\varphi}$ (in the stand-alone semantics); the plan π_i selects the service at ℓ_j; and the client and the server get involved in a new session. However, this only occurs if the history η, updated with $(\!|_\varphi$ recording the policy imposed by the client, satisfies all the policies that are active (see below for a precise definition).

Symmetrically, the rule *Close* ends a session. The client continues computing on its own, while the server H''_j is terminated. The history of the client is updated with the closing frames of *all* the policies still active in H''_j that now make no sense (computed through the auxiliary function Φ) and the closing frame of the policy φ imposed over the session.

Rule *Session* governs the independent evolution of an element within a session; and similarly, rule *Net* updates the network according to the evolution of one of its components.

Rule *Access* fires an event γ, either an access event or a policy framing; appends it to the current history η; and checks $\eta\gamma$ for validity.

The premises of rule *Synch* require a service to send/receive a message $\bar{a}/a$, and its partner to receive/send it, written as the co-action $a/\bar{a}$. The resulting communication is labelled with the (non observable) action τ. Note that a communication can only take place if both services are inside the same session.

As usual, a computation starts from the initial configuration $N_0 = \|_{j \in J} \varepsilon, H_j$, and it is a sequence $N_0 \xrightarrow{\lambda}_{\vec{\pi},R} \|_{i \in I} \eta_i, N_i \xrightarrow{\lambda'}_{\vec{\pi},R} \|_{i \in I} \eta'_i, N'_i \ldots$ Some remarks are now in order, on the way computations proceed, rather on the two ways its participants may get stuck. The first is when all the access events a service H may perform violate the security policies that are active. In this case, a resource monitor, formalised by the validity relation $\models \eta$, aborts the execution of H. Note however, that the computation proceeds if there is an event that H can fire without violating any active policies: our semantics implements the so-called *angelic* non-determinism.

The second way for deadlocking a component of a network is when two services in a session want to communicate, but the output of one of them is not matched by an input of the other, in other words, the two services are *not compliant.* Also here our semantic is angelic, in that it does not respect the requirement saying that the choice among various outputs is done regardless of the environment and of its capability of accepting the sent message.

The main task of our paper is proposing an automated technique to construct plans that drive executions with no deadlocks, namely *valid plans.* We present a static analysis, that checks the text of clients and services and guarantees that the networks, that they originate, *only* have computations that can always proceed, i.e. that at run-time a component of a network neither violates any security policies, nor does it get stuck because of missing communications. While these problems have been already studied in isolation [5,4,12,13], the combined solution we offer here is new.

We address the security issue along the lines of [5,4]. That machinery, briefly summarised below, can easily be extended to our case. Checking compliance is addressed in the next section.

3.1 Statically Checking Validity

We first intuitively define when a history $\eta \in (\mathsf{Ev} \cup \mathsf{Frm})^*$ is valid, written $\models \eta$; more detailed definitions are in [5,4]. A history η is *balanced* when either η is empty or is an event, or $\eta = (\!|_\varphi \eta' |\!)_\varphi$ with η' balanced, or $\eta = \eta'\eta''$ with both η' and η'' balanced. Hereafter, we shall only deal with histories that are prefixes of a balanced history, because such are those that show up when executing a network. Now, let $\eta^\flat$ be the history obtained by erasing all the framing events from it. For example, if $\eta_0 = \gamma\alpha(\!|_\varphi \beta|\!)_\varphi$ then $\eta_0^\flat = \gamma\alpha\beta$. A history $\eta^\flat$ respects a policy φ, in symbols $\eta^\flat \models \varphi$, if it is not recognized by the automaton φ; it is valid if it respects *all* the polices that are opened, but not closed, i.e. the policies active in η. Since our approach to security is history-dependent, we actually require that *all* the prefixes of $\eta^\flat$ respect the relevant policies. For example, consider again the history η_0 above, and let φ require that no α occurs after γ. Then, η_0 is *not* valid according to our intended meaning, because when firing β, the policy φ is activated and the prefix $\gamma\alpha$ does not obey φ — note instead that $(\!|_\varphi \gamma\ |\!)_\varphi \alpha\beta$ would be valid, as φ is no longer active after γ is fired.

The definition of validity follows, in which we use the auxiliary function $\mathcal{AP}$ for computing the multi-set of the active policies in η ($\uplus$ is multi-set union).

Validity

$$\mathcal{AP}(\varepsilon) = \emptyset \qquad \mathcal{AP}(\alpha\eta) = \mathcal{AP}(\eta)$$
$$\mathcal{AP}((\!|_\varphi \eta) = \mathcal{AP}(\eta) \uplus \{\varphi\} \qquad \mathcal{AP}(|\!)_\varphi \eta) = \mathcal{AP}(\eta) \setminus \{\varphi\}$$

A history η is *valid* ($\models \eta$) when $\forall \eta_0\eta_1$ s.t. $\eta_0\eta_1 = \eta, \varphi \in \mathcal{AP}(\eta_0).\eta_0^\flat \models \varphi$

Now the problem is verifying if all the histories generated by a given network lead to a final configuration, with no security violations. This can be done by separately checking if all its clients H are valid, i.e. that all the histories generated when it is executed are valid. Most likely, H will contain some requests, and serving them will open and eventually close possibly nested sessions with other services $H', H'', \ldots$, made available by the repository R. The idea is to suitably assemble the history expressions $H, H', H'', \ldots$, and recording in a plan for H which service to invoke for each request, so obtaining the pair $\hat{H}, \pi$. Note that $\hat{H}$ may be *non-valid*, even if the composing selected services are valid, each in isolation. Indeed, the impact on the execution history of selecting a service H_r for a request r is not confined to the execution of H_r, but it spans over the whole execution, because security is history-dependent. The validity of the composed service $\hat{H}$ depends thus on the global orchestration, i.e. on the plan π.

In order to ascertain the validity of $\hat{H}$, we resort to model checking. The history expressions $\hat{H}$ is naturally rendered as a BPA process, while finite state automata check its validity against the policies to be enforced. Because of the possible nesting of security framings, validity of history expressions is a non-regular property, so standard model checking techniques cannot be directly applied. In [5,4], a semantic-preserving transformation is presented, that removes the context-free aspects due to policy nesting: it suffices recording the opening of policies, and removing those already opened and their corresponding closures, in a stack-like fashion. In this way, (standard) model checking is efficiently feasible through specially-tailored finite state automata [5,4].

4 Checking Service Compliance

Now, we introduce a technique to construct a plan providing the assurance that also compliance between clients and services is guaranteed at run-time. Again, we can consider a client (or server) at a time, and, for each of its requests, we determine the compliant services.

First we manipulate the syntactic structure of a service in order to identify and pick up all the requests, i.e. the subterms of the form $\texttt{open}_{r,\varphi} H_1 \texttt{close}_{r,\varphi}$. Then, to check compliance of the service request r against an available service $\ell_2 : H_2$, we compute the projection of H_1 and H_2 on their communication actions. This projection removes from H_1 and H_2 all the access events α and policy opening and closing $(\!|_{\varphi}, |\!)_{\varphi}$, as well as all the *inner* service requests, i.e. the subterms of the form $\texttt{open}_{r',\varphi'} \ldots \texttt{close}_{r',\varphi}$ occurring inside H_1 and H_2. The inductive definition of the projection on communication actions follows.

Projection on Communication Actions

$$(H \cdot H')^! = H^! \cdot H'^! \qquad h^! = h \qquad \varphi(\!|H|\!)^! = H^! \qquad (\mu h.H)^! = \mu h.(H)^!$$

$$(\textstyle\sum_{i\in I} a_i.H_i)^! = \sum_{i\in I} a_i.(H_i^!) \qquad (\bigoplus_{i\in I} \overline{a}_i.H_i)^! = \bigoplus_{i\in I} \overline{a}_i.(H_i)^!$$

$$(\texttt{open}_{r,\varphi}.H.\texttt{close}_{r,\varphi})^! = \varepsilon^! = \alpha^! = \varepsilon$$

Note that if H is a closed history expression then $H^!$ is closed. Moreover the projection function $H^!$ yields a behavioural contract as defined in [12]; for this reason we feel free to call *contracts* these kind of history expressions. More precisely, the projection function produces a subset of those contracts, since in our history expressions the internal choice is always guarded by output action and the external choice is always guarded by input actions. Finally, we only have guarded tail recursion.

Because of the last restriction, it turns out that the transition system of $H^!$ is *finite state*; in other words there only is a finite number of expressions that are reachable from $H^!$ through the transitions defined by the operational semantics of history expressions in isolation.

We now recall the notion of *observable ready sets* [12]. Intuitively, these sets represent the communication actions that a service is ready to execute, so characterising the different behaviour of internal and of external choice. Roughly, a single output action at a time is offered by an internal choice, while in the external choice all the actions are available at the same time.

Definition 3 (Observable Ready Sets). *Let H be a history expression. The* observable ready set *of H is the finite set $S \subseteq \mathsf{Comm}$ given by the relation $H \Downarrow S$ inductively defined below.*

$$\varepsilon \Downarrow \emptyset \quad h \Downarrow \emptyset \qquad \bigoplus_{i\in I} \overline{a}_i.H_i \Downarrow \{\overline{a}_i\} \qquad \sum_{i\in I} a_i.H_i \Downarrow \bigcup_{i\in I}\{a_i\}$$

$$\frac{H \Downarrow S}{\mu h.H \Downarrow S} \qquad \frac{H \Downarrow S \quad S \neq \emptyset}{H \cdot H' \Downarrow S} \qquad \frac{H \Downarrow \emptyset \quad H' \Downarrow S}{H \cdot H' \Downarrow S}$$

For example, $(\overline{a_1} \oplus \overline{a_2}) \Downarrow \{\overline{a_1}\}$ and $(\overline{a_1} \oplus \overline{a_2}) \Downarrow \{\overline{a_2}\}$, while $(a_1 + a_2) \Downarrow \{a_1, a_2\}$. Also, let $H = \mu h.(\overline{a_1} \oplus \overline{a_2}) \cdot b \cdot h$, then $H \Downarrow \{\overline{a_1}\}$ and $H \Downarrow \{\overline{a_2}\}$. Moreover $\varepsilon \cdot (a+b) \cdot (\overline{d} \oplus \overline{e}) \Downarrow \{a, b\}$.

We now introduce the notion of service compliance. Given the service request $\mathtt{open}_{r,\varphi} H_1 \mathtt{close}_{r,\varphi}$ and the service H_2, we say that the two contracts $H_1^!$ and $H_2^!$ are compliant if for every possible internal action of a party, the other is able to perform the corresponding coaction. Note that the definition below does not require both parties to terminate: the client can terminate whenever all its operations have been completed. Hereafter, let $\overline{S} = \{\overline{a} | a \in S\}$.

Definition 4 (Compliance). *Two history expressions H_c and H_s are* compliant*, written $H_c \vdash H_s$, if for all $C, S \subseteq \mathsf{Comm}$, $H_1 = H_c^!$ and $H_2 = H_s^!$ are such that*

(1) $H_1 \Downarrow C$ and $H_2 \Downarrow S$ implies that $C = \emptyset$ or $C \cap \overline{S} \neq \emptyset$, and
(2) $H_1 \overset{a}{\twoheadrightarrow} H_1' \wedge H_2 \overset{co(a)}{\twoheadrightarrow} H_2'$ implies $H_1' \vdash H_2'$.

Note that the compliance relation is indeed defined in terms of the largest relation over contracts enjoying properties (1) and (2) above.

We introduce a model-checking technique for verifying if two contracts are compliant. The key idea is to reduce the problem of checking compliance to the problem of checking a safety property over a suitable finite state automaton, obtained by tailoring the notion of product automaton to contracts. Notice that this transformation will allow us to apply all the techniques and tools developed for checking safety properties.

The product automaton $\mathcal{A} = H_1^! \otimes H_2^!$ of two contracts $H_1^!$ and $H_2^!$ models the behaviour of contracts composition. Final states represent stuck configurations: these states are reached whenever the two contracts are not compliant.

Definition 5 (Product). *Let H_1' and H_2' be history expressions. The product automata $H_1 \otimes H_2$ of $(H_1')^! = H_1$ and $(H_2')^! = H_2$ is defined as follows.*
$H_1 \otimes H_2 = \langle S_1 \times S_2, \{\tau\}, \delta, \langle H_1, H_2\rangle, F\rangle$ where

- *S_1 and S_2 are the states of the transition systems of H_1 and H_2*
- *$\{\tau\}$ is the alphabet and $\langle H_1, H_2\rangle$ is the initial state*
- *the transition function δ is:*

$$\delta = \{(\langle H_1, H_2\rangle, \tau, \langle H_1', H_2'\rangle) | H_1 \overset{a}{\twoheadrightarrow} H_1' \wedge H_2 \overset{co(a)}{\twoheadrightarrow} H_2' \wedge \langle H_1, H_2\rangle \notin F\}$$

– *the set of final states is* $F = \{\langle H_1, H_2\rangle | H_1 \neq \varepsilon \wedge \neg(i) \vee \neg(ii)\}$ *where :*
  (i) $\exists \overline{a}.(H_1 \overset{\overline{a}}{\twoheadrightarrow} H_1' \vee H_2 \overset{\overline{a}}{\twoheadrightarrow} H_2')$
  (ii) $(\forall H_1 \overset{\overline{a}}{\twoheadrightarrow} H_1', \exists H_2 \overset{a}{\twoheadrightarrow} H_2') \wedge (\forall H_2 \overset{\overline{a}}{\twoheadrightarrow} H_2', \exists H_1 \overset{a}{\twoheadrightarrow} H_1')$

The correctness of the definition of the product automaton relies on the fact that the projection function yields finite state contracts since recursive behaviour is obtained only via tail recursion.

Note that condition (i) ensures that both services are not waiting on input actions. Condition (ii), instead, ensures that for all the possible output actions that a service is ready to fire, the other party is ready to perform the corresponding input action. Intuitively, H_1 and H_2 are compliant if and only if the language of the product automaton $\mathcal{A}$ is empty, no final states exist in which the above conditions do not hold.

Lemma 1. *Let H_c and H_s be closed history expressions with $H_1 = H_c^!$ and $H_2 = H_s^!$. For all $C, S \subseteq \mathsf{Comm}$ such that $H_1 \Downarrow C$, $H_2 \Downarrow S$ we have that $C \cap \overline{S} \neq \emptyset$ if and only if conditions (i) and (ii) of Definition 5 hold.*

Proof. $(\rightarrow)$ We know that either H_1 or H_2 performs an output action, so condition (i) holds. W.l.o.g. assume that H_1 performs an output. The proof in the other case is symmetric. By definition of observable ready sets and $C \cap \overline{S} \neq \emptyset$, we have that H_1 must be of the form $H \cdot H'$ with H either $\bigoplus_{i\in I} \overline{a}_i.H_i$ or $\mu h. \bigoplus_{i\in I} \overline{a}_i.H_i$, (recall that $\varepsilon \cdot H \equiv H \equiv H \cdot \varepsilon$), with $I \neq \emptyset$. Also H_2 must be of the form $H_2' \cdot H_2''$ with H_2' either $\sum_{j\in J} a_j.H_j$ or $\mu h. \sum_{j\in J} a_j.H_j$ with $|J| \neq \emptyset$. For H_1 we have $|I|$ different ready sets forming $IC = \{\{\overline{a_i}\} | H_1 \Downarrow \{\overline{a_i}\}\}$. Instead H_2 has a single ready set of the form $S = \{a_j | j \in J\}$. Now by hypothesis $\forall C \in IC$. $C \cap \overline{S} \neq \emptyset$, that implies $(\forall H_1 \overset{\overline{a}}{\twoheadrightarrow} H_1', \exists H_2 \overset{a}{\twoheadrightarrow} H_2') \wedge (\forall H_2 \overset{\overline{a}}{\twoheadrightarrow} H_2', \exists H_1 \overset{a}{\twoheadrightarrow} H_1')$ because H_2 performs no output actions.

$(\leftarrow)$ By (i) we have that either C or S contain an output action. W.l.o.g. assume that H_1 performs an output. The proof in the other case is symmetric. By definition of operational semantics of history expressions, H_1 must be of the form $H \cdot H'$ with H either $\bigoplus_{i\in I} \overline{a}_i.H_i$ or $\mu h. \bigoplus_{i\in I} \overline{a}_i.H_i$ with $I \neq \emptyset$. Hence by definition H_1 has $|I|$ different ready sets in $IC = \{\{\overline{a_i}\} | H_1 \Downarrow \{\overline{a_i}\}\}$. By condition (ii), H_2 must be able to execute any corresponding coaction. Therefore by definition of operational semantic of history expressions, H_2 must be of the form $H_2' \cdot H_2''$ with H_2' either $\sum_{j\in J} a_j.H_j$ or $\mu h. \sum_{j\in J} a_j.H_j$ with $|J| \neq \emptyset$. Hence by definition H_2 has a single ready set of the form $S = \{a_j | j \in J\}$. Condition (ii) ensures that $\forall C \in IC$ it must be $\overline{C} \cap S \neq \emptyset$. □

We now state our main theorem that guarantees compliance of two services whenever their languages have an empty intersection. Its proof is omitted because of lack of space; the interested reader can find it on the web page `www.di.unipi.it/user/basile/papers/pact2013full.pdf`.

Theorem 1 (Compliance)
Let H_1' and H_2' be closed history expressions and let $(H_1')^! = H_1$ and $(H_2')^! = H_2$. Then $H_1 \vdash H_2$ if and only if $L(H_1 \otimes H_2) = \emptyset$.

An important consequence of our model-checking technique is that the property of progress of a session (even for infinite executions) is not a liveness property, but an invariant property [2]. An invariant property P_{inv} only inspects a state at time, without looking at all the past history, i.e. $P_{inv} = \{s_0 s_1 s_2 \ldots | \forall j \geq 0. s_j \models \Phi\}$, where $s_0 s_1 s_2 \ldots$ is a history. Since conditions (i) and (ii) of Definition 5 do not inspect the past states, it turns out that compliance is an invariant property: a subset of the safety properties.

Theorem 2. *Compliance is an invariant property.*

Proof. Given two services $(H_1')^! = H_1$ and $(H_2')^! = H_2$ by Theorem 1 $H_1 \vdash H_2$ iff $L(H_1 \otimes H_2) = \emptyset$, i.e. all states $\langle H_1', H_2' \rangle$ reachable from $\langle H_1, H_2 \rangle$ are such that $\langle H_1', H_2' \rangle \models (H_1' = \varepsilon \vee (i) \wedge (ii))$ ((i) and (ii) conditions of Definition 5). □

Since all invariant properties are safety properties, the following corollary holds.

Corollary 1. *Compliance is a safety property.*

5 Verifying Services Secure and Unfailing

We have all the means to statically verify whether a network of services will evolve with neither security nor compliance violations. Given a repository R and a vector of clients, pick up one of them, say H, at a time; generate a valid plan π_H for H; for each request $\texttt{open}_{r,\varphi} H_1 \texttt{close}_{r,\varphi}$ occurring in the composed service check if $H_1 \vdash H_2$, where $\pi_H(r) = \ell_2$ and $\ell_2 \in R$. If all these steps succeed, switch off any run-time monitor, and live happily: nothing bad will happen.

This result relies on suitable extensions of the methodology proposed in [4], and on a careful definition of service sessions, possibly nested, and of compliance. Indeed, Theorem 2 shows that compliance is a safety property, so paving the way to its verification via standard model-checking, with existing tools [7].

As a matter of fact, our work establishes a novel connection between the world of service contracts and the world of security. We plan to extend our approach in some directions. A first is for modelling more carefully the availability of services, that now can replicate themselves unboundedly many times. We are confident that more detailed rules for opening and closing sessions can be easily given. A major line of research concerns extending our verification methodology to include quantitative information in the security policies, along the lines of [14]. We would like also to modify the model checker and related tools of [7] so to completely mechanise our proposal.

References

1. Artikis, A., Sergot, M.J., Pitt, J.V.: Specifying norm-governed computational societies. ACM Trans. Comput. Log. 10(1) (2009)
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
3. Bartoletti, M.: Usage automata. In: Degano, P., Viganò, L. (eds.) ARSPA-WITS 2009. LNCS, vol. 5511, pp. 52–69. Springer, Heidelberg (2009)

4. Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. Journal of Computer Security 17(5), 799–837 (2009)
5. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Call-by-contract for service discovery, orchestration and recovery. In: Wirsing, M., Hölzl, M. (eds.) SENSORIA. LNCS, vol. 6582, pp. 232–261. Springer, Heidelberg (2011)
6. Bartoletti, M., Tuosto, E., Zunino, R.: On the realizability of contracts in dishonest systems. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 245–260. Springer, Heidelberg (2012)
7. Bartoletti, M., Zunino, R.: LocUsT: a tool for checking usage policies. Tech. Rep. TR-08-07, Dip. Informatica, Univ. Pisa (2008)
8. Bartoletti, M., Zunino, R.: A calculus of contracting processes. In: LICS, pp. 332–341. IEEE Computer Society (2010)
9. Buscemi, M.G., Montanari, U.: CC-pi: A constraint-based language for specifying service level agreements. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 18–32. Springer, Heidelberg (2007)
10. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006)
11. Carpineti, S., Laneve, C.: A basic contract language for web services. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 197–213. Springer, Heidelberg (2006)
12. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. 31(5) (2009)
13. Castagna, G., Padovani, L.: Contracts for mobile processes. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 211–228. Springer, Heidelberg (2009)
14. Degano, P., Ferrari, G.-L., Mezzetti, G.: On quantitative security policies. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 23–39. Springer, Heidelberg (2011)
15. Reisig, W.: Petri Nets: An Introduction, Monographs in Theoretical Computer Science. An EATCS Series, vol. 4. Springer (1985)

FuPerMod: A Framework for Optimal Data Partitioning for Parallel Scientific Applications on Dedicated Heterogeneous HPC Platforms

David Clarke, Ziming Zhong, Vladimir Rychkov, and Alexey Lastovetsky

School of Computer Science and Informatics, University College Dublin,
Belfield, Dublin 4, Ireland
{david.clarke,ziming.zhong}@ucdconnect.ie,
{vladimir.rychkov,alexey.lastovetsky}@ucd.ie
http://hcl.ucd.ie

Abstract. Optimisation of data-parallel scientific applications for modern HPC platforms is challenging in terms of efficient use of heterogeneous hardware and software. It requires partitioning the computations in proportion to the speeds of computing devices. Implementation of data partitioning algorithms based on computation performance models is not trivial. It requires accurate and efficient benchmarking of devices, which may share the same resources but execute different codes, appropriate interpolation methods to predict performance, and mathematical methods to solve the data partitioning problem. In this paper, we present a software framework that addresses these issues and automates the main steps of data partitioning. We demonstrate how it can be used to optimise data-parallel applications for modern heterogeneous HPC platforms.

Keywords: heterogeneous computing, data partitioning, computation performance models, hybrid platforms.

1 Introduction

Many scientific applications implement data-parallel algorithms, originally designed for homogeneous HPC platforms. The applications range from linear algebra routines to computer simulations, such as computational fluid dynamics. They are characterised by divisible computational workload, which is directly proportional to the size of data and dependent on data locality. In order to execute data-parallel scientific applications on a highly heterogeneous HPC platforms efficiently, computational workload has to be distributed between computing devices in proportion to their speeds. Our target architecture is a dedicated highly heterogeneous HPC platform, which has a stable performance in time, a complex hierarchy of heterogeneous computing devices, and a heterogeneous software stack. We consider this platform as a hierarchical heterogeneous distributed-memory system, and therefore, apply data partitioning, a method of load balancing widely used for distributed-memory supercomputers.

Data partitioning algorithms use performance models of computing devices and computation kernels to distribute workload. In this work, we address the problem of implementation of data partitioning algorithms in data-parallel applications for dedicated heterogeneous platforms.

Data partitioning algorithms based on computation performance models require accurate and efficient performance measurement, implementation of interpolation methods for realistic performance prediction, and formalisation and solution of the data partitioning problem. For static data partitioning, the optimality of the load distribution is critical, while for dynamic data partitioning, the cost-efficiency is equally important. In the first case, the use of exhaustive benchmarks to build very detailed computation performance models in advance is justified. In the second case, only short measurements can be performed, whose inaccuracy has to be compensated by advanced interpolation methods. On modern multicore and hardware-accelerated platforms, special performance measurement techniques and computation performance models are required to take into account resource contention. Despite the active research in the area of data partitioning, there is no software available that would address these challenges. In this paper, we present such a software framework.

Our software framework is designed to construct computation performance models for any data-parallel application to a given accuracy and cost-effectiveness. There are two types of models supported: constant and functional. The models can be built either in advance to be used in static data partitioning, or at runtime during dynamic load balancing. The framework provides a range of general-purpose data partitioning algorithms based on computation performance models. The choice of algorithms is determined by the user's applications. In this paper, we demonstrate how they can be used for optimal execution of the parallel matrix multiplication and the Jacobi method. The framework supports a wide range of dedicated heterogeneous platforms consisting of uniprocessors, multicores and hardware accelerators. The framework is extensible; new measurement techniques for new types of hardware can be added and other computation performance models and data partitioning algorithms can be included.

The rest of this paper is organised as follows. In Section 2, we overview existing data partitioning software. In Section 3, we discuss the main challenges in optimisation of data-parallel applications for heterogeneous platforms, and formulate the features of a framework for data partitioning based on computation performance models. In Section 4, we present the new software framework and describe the use cases, namely optimisation of heterogeneous parallel matrix multiplication and dynamic load balancing of the Jacobi method.

2 Existing Data Partitioning Software

Matrices and meshes are the most common objects of parallel scientific applications. Since they can be represented as graphs, most data partitioning software implement graph partitioning algorithms. Algorithms implemented in ParMetis [9], SCOTCH [4], JOSTLE [16] reduce the number of edges between the target subdomains, and hence, minimise the total communication cost of the application. They take into account heterogeneity of the platform by specifying

– weights of the target subdomains, which represent the relative speeds of processors [9], or
– a weighted graph of the platform, which contains information about the speeds of processors and the bandwidths of links [4,16].

Algorithms implemented in Zoltan [3], PaGrid [1] minimise the execution time of the application using some cost function. The cost function depends on both the graph and the parameters of the heterogeneous platform defined by:

– a description of the hierarchy of processors [3], or
– a weighted graph of the platform, with the speeds of processors and the latencies/bandwidths of links [1].

To distribute computations between the processors, all these graph partitioning libraries use simplistic computation performance models, where the speeds of processors are given by constants (weights). Despite the fact that the result of data partitioning is very sensitive to the weights, the libraries do not provide any methods to find the values that balance the load for given data-parallel application on heterogeneous platform. Application programmers are responsible for building the computation performance models and distributing the load.

Traditionally, the constants characterising the performance of the processors are found as their relative speeds demonstrated during the execution of a serial benchmark code solving locally the core computational task of some given size. This approach is not always accurate and may result in non-optimal partitioning on modern highly heterogeneous platforms as it was demonstrated in [6]. Existing data partitioning software, which is based on this approach, do not take into account memory hierarchy, hierarchy of computing devices, software heterogeneity, optimisations and out-of-core techniques used in software.

For modern heterogeneous platforms, more realistic computation performance models have been proposed along with more elaborate general-purpose model-based data partitioning algorithms to find the optimal load distribution ratios, which can be used as weights in graph partitioning. However, integration of these algorithms into data-parallel applications is not trivial. In the following section, we discuss the main challenges of software implementation of heterogeneous data-parallel applications and define the features of a framework for data partitioning based on computation performance models.

3 Optimisation of Data-Parallel Applications for Heterogeneous Platforms

In this section, we analyse the main challenges application programmers face while optimising data-parallel applications for modern heterogeneous HPC architectures. Given a data-parallel scientific application, originally designed for distributed-memory platforms and implemented with help of MPI, how to execute it efficiently on a heterogeneous platform? The total volume of communications is minimised at the application level (for example, by multilevel graph partitioning in mesh applications [9], or by arrangement of matrix blocks in matrix applications [2]). In the computationally intensive part, the application calls a library of routines, for which the hardware-optimised implementations are available (for example, multi-threaded and GPU solvers). Therefore, in order to execute this application efficiently on the heterogeneous platform, we do not design new hybrid kernels that employ multiple computing devices simultaneously. Instead, we use existing high performance kernels for these devices and distribute the application data unevenly between them, based on the a priori information about their performance.

A general-purpose data partitioning algorithms based on computation performance models proceeds as follows. As input, it requires performance models, which can be constructed either in advance or at run-time. The models approximate the speed of the application on each of the computing devices, and in their turn require empirical information about the real performance. This information can be obtained from the benchmarks that assess the performance of the application on the devices. Therefore, the main challenges the application programmer faces are accurate and efficient performance measurement, construction of computation performance models and implementation of model-based data partitioning algorithms.

Accurate and cost-effective methods of performance measurement are paramount for data partitioning to work in real-life heterogeneous environments. The use of wrong estimates can fully destroy the resulting performance of the application. Performance can be found by benchmarking *a computation kernel*, a serial code performing much less computations but still representative for the entire application [10]. For example, computationally intensive applications often perform the same core computation multiple times in a loop. The benchmark made of one such core computation can be representative of the performance of the whole application and can be used as a kernel. **Timing the computation kernel on heterogeneous devices** may be non-trivial, and therefore, its automation would facilitate development of data-parallel applications, especially on the platforms where special techniques are required for accurate performance measurements, such as multicore and hardware-accelerated platforms.

For example, on multicore platforms, parallel processes interfere with each other through shared memory so that the speed of individual cores cannot be measured independently. In this case, the performance of cores in a group can be measured, when all cores are executing the benchmarks in parallel [18]. Interactions between CPUs and GPUs include data transfers between the host and GPU memory over PCI Express, launching of GPU kernels, and some other operations. Performance measurement techniques for heterogeneous GPU-accelerated systems were studied in [13]. It was concluded that the synchronous approach, when the host CPU core observes the beginning and the end of an operation, is valid for measurement of routines implemented in synchronous libraries, such as CUBLAS. This technique covers all interactions between devices and does not require any special measurement mechanisms. The performance of out-of-core routines can also be measured from the host CPU core. Incorporation of these **performance measurement techniques** into the data-parallel application add extra complexity; they have to be implemented as routine procedures.

The results of performance measurements are used in computation performance models, which implement different interpolation methods to predict the execution time and speed. On this prediction, heterogeneous data partitioning algorithms will be based. A number of models and algorithms have been proposed. Their choice depends on the data-parallel application and the heterogeneous platform. The software framework for data partitioning has to provide a collection of such models and algorithms. We briefly summarise the applicability of recent work related to data partitioning on heterogeneous multicore and multi-GPU platforms.

When the problems fit the main memory of the processors/devices and the processors/devices execute the same codes for these problems, the absolute speeds do not vary. In this case, data partitioning algorithms based on **constant performance model** (CPM) can be used. In [8], constants representing the sustained performance of the application on CPU/GPU were used to partition data. The constants were found a priori. In [17], a similar constant performance model was proposed, but it was built adaptively, using the history of performance measurements. CPM-based algorithms are cost efficient and do not introduce much complexity into heterogeneous applications. The fundamental assumption of these algorithms is that the absolute speed of processors/devices does not depend on the size of a computational task. However, it becomes less accurate when the partitioning of the problem results in some tasks fitting into different levels of memory hierarchy (i), or when processors/devices switch between different codes to solve the same computational problem (ii).

For the cases (i) and (ii), more elaborate computation performance models and data-partitioning algorithms have been proposed. In [12], the execution time of CPU/GPU was approximated by linear functions of problem size, and an empirical approach to estimate the application-specific **linear performance models** was presented. A more elaborate analytical predictive model was proposed in [14]; it is also application-specific, however, not only the values of parameters, but also their number and the predictive formulas are defined individually for each application, based on thorough performance analysis of the main steps of the application. In [14], it was admitted that linear models might not fit the actual performance in the case of resource contention (iii), and therefore, they were replaced by the **analytical piecewise model**. This model can achieve high accuracy but there is no generic way to build it for an arbitrary application and hardware. Nevertheless, analytical models and model-based data partitioning algorithms can be implemented once per class of applications and can be included into a framework for data partitioning.

For an arbitrary data-parallel application to be executed for a wide range of problem sizes on a platform with highly heterogeneous hardware/software and resource contention, we proposed the **functional performance model** (FPM), where the speed is represented by a function of problem size that is built empirically and integrates performance characteristics of both the architecture and the application [10]. Under the functional performance model, the speed of each processor is represented by a continuous function of the problem size. The speed is defined as the number of *computation units* processed per second. Such a performance model is application and hardware specific. In particular, this means that the computation unit can be defined differently for different applications. The important requirement is that the computation unit does not have to vary during the execution of the application. This model can be estimated in the same way for any data-parallel application and applicable in situations (i)-(ii). It approximates the execution time and speed using piecewise linear or Akima spline interpolation [15]. Originally, the functional performance model was designed for uniprocessor machines: it provided optimal data partitioning [7] and efficient dynamic load balancing [6] on heterogeneous networks of uniprocessors. Later, this approach was extended to multicore [18] and hybrid CPU/GPU [19] platforms. Taking into account resource contention, situation (iii), we introduced *a speed of multiple cores*, when multiple cores simultaneously execute the same computation kernel, and *a combined speed of a GPU and a dedicated host CPU*, when the GPU executes a computation kernel and the CPU provides memory management. Integration of all these features into a data-parallel application is challenging and requires appropriate software implementation.

Elaborate computation performance models provide more accurate prediction but complicate data partitioning algorithms. In contrast to the traditional data partitioning algorithms, which distribute computations in proportion to constant speeds, the algorithms based on functional models require solving a system of equations whose solution yields the balance. If the speeds are defined by predictive formulas, the solution of the load balancing problem can be found analytically [12], [14]. Otherwise, if the speeds are interpolated from empirical data, like in [10], the solution can be found geometrically [10] or numerically [15]. Implementation of model-based data partitioning from scratch within a heterogeneous data-parallel application is challenging due to **complexity of data partitioning algorithms**. Therefore, routines implementing these algorithms forms a key functionality of a framework facilitating development of applications for heterogeneous platforms.

In this paper, we present the software framework that addresses the above challenges. On several examples, we illustrate how to adapt data-parallel MPI applications to hybrid heterogeneous platforms, using this framework.

4 New Framework for Model-Based Data Partitioning

In this section, we give a high-level outline of the new framework for model-based data partitioning FuPerMod, available through the open-source license from `http://hcl.ucd.ie/project/fupermod`. The framework provides the programming interface for:

- accurate and cost-effective performance measurement,
- construction of computation performance models implementing different methods of interpolation of time and speed,

– invocation of model-based data partitioning algorithms for static and dynamic load balancing.

This functionality can be incorporated into a data-parallel applications as follows. First, the application programmer has to provide the serial code for the computation kernel of their application and define its computation unit by using the API provided. This code will be used for computation performance measurements, which can be carried out either within the application or separately, in order to obtain the a priori performance information. Then, the programmer chooses the appropriate computation performance model and data partitioning algorithm, and incorporates them into the application. Upon execution of the data-parallel application on the heterogeneous platform, the models of processors/devices will be constructed and the data partitioning algorithm will yield the optimal distribution of workload for a given problem size. Finally, the programmer is responsible for distribution of the application data according to the optimum distribution given in computation units.

4.1 Computation Performance Measurement

The programming interface for computation performance measurement consists of a data structure encapsulating the computation kernel, *fupermod_kernel*, the benchmark function, *fupermod_benchmark*, and a data structure storing the result of the measurement, *fupermod_point*.

The serial code of the computation kernel has to be provided together with the functions to allocate and deallocate the data for a problem size given in computation units. In these functions, the application programmer defines the computation unit and reproduces the memory requirements of the application. To enable conversion of speed from units/sec to FLOPS, the programmer has to specify the complexity of the computation unit. As a whole, *fupermod_kernel* has the following interface:

```
struct fupermod_kernel {
  double (*complexity)(int d, void* params);
  int (*initialize)(int d, void* params);
  int (*execute)(pthread_mutex_t* mutex, void* params);
  int (*finalize)(void* params);
};
```

– *complexity* is a pointer to the function that returns the complexity of computing d units;
– *initialize/finalize* allocate and deallocate memory for the problem of d computation units (create and destroy the execution context for the kernel);
– *execute* executes the computation kernel in a separate thread;
– *params* stores the execution context of the kernel;
– *mutex* protects some resources, when kernel is terminated during a long run.

Let us consider how to define the computation kernel for a typical data-parallel application, such as matrix multiplication.

In this application, square matrices A, B and C are partitioned over a 2D arrangement of heterogeneous processors so that the area of each rectangle is proportional to the speed of the processor that handles the rectangle. This speed is given by the speed function of the processor for the assigned problem size. Figure 1(a) shows one iteration of matrix multiplication, with the blocking factor b parameter, adjusting the granularity of communications and computations [5]. At each iteration of the main loop, pivot column of matrix A and pivot row of matrix B are broadcasted horizontally and vertically, and then matrix C is updated in parallel by the GEMM routine of the Basic Linear Algebra Subprograms (BLAS). In this application, we use the matrix partitioning algorithm [2] that arranges the submatrices to be as square as possible, minimising the total volume of communications and balancing the computations on the heterogeneous processors.

We assume that the total execution time of the application can be approximated by multiplying the execution time of a single run of the computational kernel by the number of iterations of the application. Therefore, the speed of the application can be estimated more efficiently by measuring just one run of the kernel. For this application, the computation kernel on the processor i will be an update of a $b \times b$ block of the submatrix C_i with the parts of pivot column $A_{(b)}$ and pivot row $B_{(b)}$: $C_i+ = A_{(b)} \times B_{(b)}$ (Fig. 1(b)). This block update represents one computation unit of the application. The processor i is to process $m_i \times n_i$ such computation units, which is equal to the area of the submatrix if measured in blocks. For nearly-square submatrices, which is the case in this application, one parameter, area d_i, can be used as a problem size.

Therefore, in the *initialize* function, for the problem size d_i, we define $m_i = \lfloor\sqrt{d_i}\rfloor; n_i = \lfloor d_i/m_i\rfloor$. We allocate and initialise $(m_i \times b) \times (n_i \times b)$ elements for each of the submatrices A_i, B_i and C_i. We allocate the working buffers $A_{(b)}$ and $B_{(b)}$ of sizes $(m_i \times b) \times b$ and $b \times (n_i \times b)$ respectively. The *execute* function for this kernel will be representative of the local work performed by one iteration of the main loop of the application. To replicate the local overhead of the MPI communication it does a memory copy from part of submatrices A_i and B_i to working buffers $A_{(b)}$ and $B_{(b)}$ respectively. It then calls the GEMM routine once with $A_{(b)}$, $B_{(b)}$ and C_i. Having the same memory access pattern as the

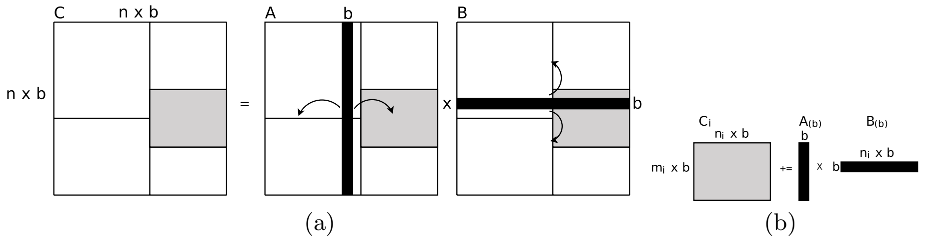


Fig. 1. Heterogeneous parallel column-based matrix multiplication (a) and its computational kernel (b)

whole application, the kernel will be executed at nearly the same speed as the whole application. The *complexity* function returns the number of arithmetic operations performed by the kernel: $2 \times (m_i \times b) \times (n_i \times b) \times b$.

Performance measurement of this kernel on heterogeneous devices that share resources and use different programming models is challenging. In our previous work, we proposed the measurement techniques for a multicore node [18] or GPU-accelerated node [19], which are now implemented in the FuPerMod framework. They provide reproducible results within some accuracy and can be summarised as follows. Automatic rearranging of the processes provided by operating system may result in performance degradation, therefore, we bind processes to cores to ensure a stable performance. Then, we synchronise the processes that share resources (on a node or a socket), in order to minimise the idle computational cycles, aiming at the highest floating point rate for the application. Synchronisation also ensures that the resources will be shared between the maximum number of processes, generating the highest memory traffic. To ensure the reliability of the measurement, experiments are repeated multiple times until the results are statistically correct.

GPU depends on a host process, which handles data transfer between the host and device and launches kernels on the device. A CPU core is usually dedicated to deal with the GPU, and can undertake partial computations simultaneously with the GPU. Therefore, we measure the combined performance of the dedicated core and GPU, including the overhead incurred by data transfer between them. Due to limited GPU memory, the execution time of GPU kernels can be measured only within some range of problem sizes, unless out-of-core implementations, which address this limitation, are available.

To measure the performance of a computation kernel on heterogeneous processors/devices, FuPerMod provides a function *fupermod_benchmark*, which has the following interface:

```
int fupermod_benchmark(                 struct fupermod_point {
  fupermod_kernel* kernel, int d,         int d;
  fupermod_precision precision,           double t;
  MPI_Comm comm_sync,                     int reps;
  fupermod_point* point                   double ci;
);                                      };
```

This function initialises the *kernel* for the problem size *d* and executes it multiple times accordingly to the *precision* argument, which defines the number of repetitions and statistical parameters. The kernel can be executed in multiple processes. MPI communicator *comm_sync* is used to synchronise the processes running on a multi-CPU/GPU node. The function returns a *point*, which contains the results of the measurement: the problem size in computation units, *d*; the measured execution time, *t*; the number of repetitions the measurement has actually taken, *reps*; and the confidence interval of the measurement, *ci*. Arrays of these experimental points are then used to model the performance of CPU core(s), or the bundled performance of a GPU and its dedicated CPU core, or the total performance of a multi-CPU/GPU node.

4.2 Computation Performance Models

The key abstraction of the programming interface for computation performance modeling is *fupermod_model*, which has the following interface:

```
struct fupermod_model {
  int count;
  fupermod_point* points;
  double (*t)(fupermod_model* model, double x);
  int (*update)(fupermod_model* model, fupermod_point point);
};
```

It encapsulates experimental points obtained from measurements, which are given by the *count* and *points* data fields, and the approximation of the time function, *t*. *update* specifies how the approximation changes after adding a new experimental point. The speed in FLOPS is evaluated using the approximated time and the complexity of the computation kernel: $s(x) = complexity(x)/time(x)$, where x is a problem size given in computation units. These approximations are used in the model-based data partitioning algorithms to predict the computation performance and distribute the workload proportionally.

Currently, FuPerMod implements the following performance models:

- CPM (requires only one experimental point);
- FPM based on the piecewise linear interpolation of the time function;
- FPM based on the Akima spline interpolation of the time function.

The first FPM is based on some assumptions on the shape of the speed function [10]. In addition to the piecewise linear interpolation, it coarsens the real performance data in order to satisfy those assumptions, as shown in Fig. 2(a).

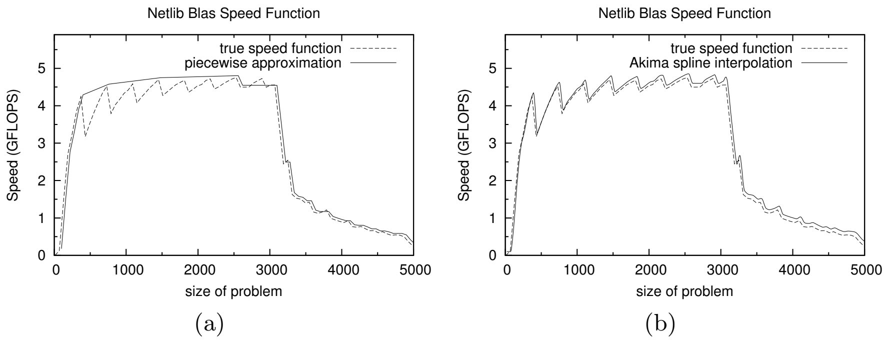


Fig. 2. Speed functions of the matrix multiplication kernel based on the Netlib BLAS GEMM: (a) piecewise linear interpolation, (b) Akima spline interpolation

The FPM based on the Akima spline interpolation removes these restrictions [15], and therefore, represents the speed of the processor with more accurate continuous functions (Fig. 2(b)). The *fupermod_model* data structure can be used to implement other computation performance models, for example, application-specific analytical models, such as [14].

4.3 Static Data Partitioning

Computation performance models of processes are used as input for model-based data partitioning algorithms. The FuPerMod framework currently provides the following algorithms:

– basic algorithm based on CPMs;
– geometrical algorithm based on the piecewise-linear FPMs;
– numerical algorithm based on the Akima-spline FPMs.

The CPM-based algorithm divides the data in proportion to the constant speeds. This is the fastest but least accurate data partitioning algorithm. It is appropriate for the cases when it has been observed that the speeds do not vary much. The geometrical algorithm implements iterative bisection of the speed functions with lines passing through the origin of the coordinate system [10]. Convergence of this algorithm is ensured by putting restrictions on the shape of the speed functions, which is implemented in the piecewise-linear FPMs. The numerical algorithm applies multidimensional solvers to numerical solution of the system of non-linear equations that formalise the problem of optimal data partitioning [15]. It can be applied to smooth speed functions of any shape. As input, the algorithm takes the Akima-spline FPMs, since this approximation provides continuous derivative.

Data partitioning algorithms have the following interface:

```
typedef int (*fupermod_partition)(
  int size, fupermod_model** models, fupermod_dist* dist);
```

where *size* is the number of the processes, *models* is an array of the models corresponding to the processes, and *dist* is the distribution of data. The distribution is an input/output argument and has the following structure:

```
struct fupermod_dist {                struct fupermod_part {
  int D;                                int d;
  int size;                             double t;
  fupermod_part* parts;               };
};
```

where D is the total problem size to partition (in computation units); *size* is the number of processes; *parts* is the array specifying the workload d that will be assigned to the processes, and the predicted computing time t of the workload. After execution of the data partitioning algorithm, the application programmer distributes the workload in accordance with the *dist* argument. A sample code demonstrating how to use the programming interface for data partitioning will be provided below, within a more practical example of dynamic load balancing.

The cost of experimentally building a full computation performance model, i.e. a functional model for the full range of problem sizes, may be very high, which limits the applicability of the above partitioning algorithms to situations where the construction of the models and their use in the application can be separated. For example, if we develop an application that will be executed on the same platform multiple times, we can build the full models once and then use these models multiple times during the repeated execution of the application. In this case, the time of construction of the models can become very small compared to the accumulated performance gains during the multiple executions of the optimized application. Building full functional performance models is not suitable for an application that is run a small number of times on a platform. In this case, computations should be optimally distributed between processors without *a priori* information about execution characteristics of the application running on the platform. In the following section, we describe the programming interface for dynamic data partitioning and load balancing, which can be used to design applications that automatically adapt at runtime to any set of heterogeneous processors.

4.4 Dynamic Data Partitioning and Load Balancing

FuPerMod provides the efficient data partitioning algorithms that do not require performance models as input. Instead, they approximate the speeds around the relevant problem sizes, for which performance measurements are made during the execution of the algorithms. These algorithms do not construct complete performance models, but rather partially estimate them, sufficiently for optimal distribution of computations. They balance the load not perfectly, with a given accuracy. The low execution cost of these algorithms makes them suitable for employment in self-adaptable applications. Currently, FuPerMod provides two such algorithms, designed for dynamic data partitioning [11] and dynamic load balancing [6].

The dynamic algorithms perform data partitioning iteratively, using the partial estimates instead of the full computation performance models. At each iteration, the solution of the data partitioning problem gives new relevant problem sizes. The performance is measured for these problem sizes, and the partial estimates are refined. In the case of dynamic data partitioning, the measurements are made by benchmarking the representative computation kernel of the application. In the case of dynamic load balancing, the real execution of one iteration of the application is timed. Figure 3 shows a few steps of dynamic data partitioning for piecewise linear FPMs and geometrical data partitioning algorithm.

The programming interface for the dynamic algorithms consists of a data structure *fupermod_dynamic*, specifying the context of their execution, and two

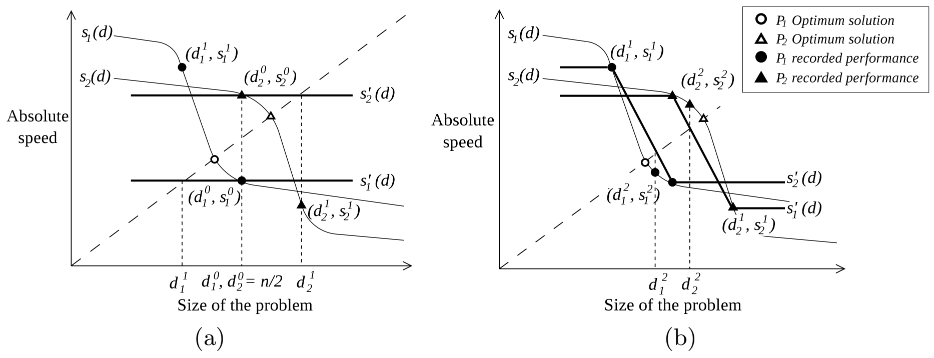


Fig. 3. Construction of the partial FPMs based on piecewise linear interpolation, using the geometrical data partitioning algorithm

functions *fupermod_partition_iterate* and *fupermod_balance_iterate*, implementing one step of dynamic partitioning and load balancing respectively:

```
struct fupermod_dynamic {
  fupermod_partition partition;
  int size;
  fupermod_model** models;
  fupermod_dist* dist;
}
int fupermod_partition_iterate(fupermod_dynamic*, MPI_Comm comm,
  fupermod_precision precision, fupermod_benchmark* benchmark,
  double eps);
int fupermod_balance_iterate(fupermod_dynamic*, MPI_Comm comm,
  struct timespec start);
```

The context includes the pointer to a data partitioning algorithm, *partition*, current partial estimates, *models*, and near-optimal data partition, *dist*. Both function invoke the data partitioning algorithm once, using the current estimates, and store the result in *dist*. The dynamic data partitioning function performs the *benchmark*, with the statistical parameters *precision*, while the dynamic load balancing function uses the *start* time of the current iteration of the application to time. Then both function update the partial estimates. The dynamic data partitioning also requires the accuracy, *eps*, as a termination criterion.

In conclusion, we demonstrate how to use this API for optimisation of another data-parallel application, which implements the Jacobi method. This application distributes the matrix and vectors by rows between the processors and iteratively solves the system of equations. In the source code below, the partial FPMs based on piecewise linear interpolation are constructed at runtime during the iterations of the Jacobi method. At each iteration, the load balancing function invokes the geometrical data partitioning algorithm. The system of equations is redistributed accordingly to the newly obtained data distribution. Figure 4 demonstrates that after several iterations of the application, the load is balanced.

```
MPI_Comm_size(comm, &size);
// FPMs based on piecewise linear interpolation
fupermod_model** models = malloc(sizeof(fupermod_model*) * size);
for (i = 0; i < size; i++)
  models[i] = fupermod_model_piecewise_alloc();
// context for dynamic load balancing
fupermod_dynamic balancer = { fupermod_partition_geometric,
  size, models, fupermod_dist_alloc(D, size) };
// current distribution, initially even
fupermod_dist* dist = fupermod_dist_alloc(D, size);
// Jacobi data: dist->parts[i].d rows of matrix and vectors
double *A, *b, *x; // allocation, initialisation
// main loop
double error = DBL_MAX;
while (error > eps) {
  // redistribution of Jacobi data accordingly to balancer.dist
  jacobi_redistribute(comm, dist, A, b, x, balancer.dist);
  // store the current distribution
  fupermod_dist_copy(dist, balancer.dist);
  struct timeval start;
  gettimeofday(&start, NULL);
  // Jacobi iteration
  jacobi_iterate(comm, dist, A, b, x, &error);
  // load balancing with the (dist->parts[i].d, now-start) point
  fupermod_balance_iterate(&balancer, comm, start);
}
```

In this paper, we presented a framework for general-purpose data partitioning based on computation performance models. This framework provides a range of algorithms and models for optimisation of different data-parallel scientific applications on modern heterogeneous platforms.

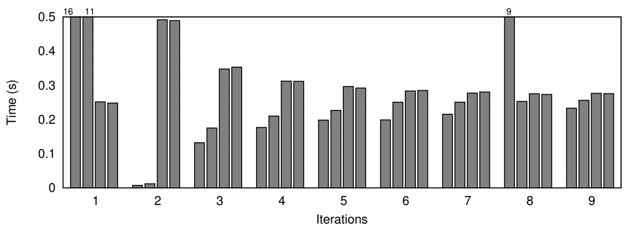


Fig. 4. Dynamic load balancing of Jacobi method with geometrical data partitioning

Acknowledgment. This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/IN.1/I2054. Experiments were carried out on Grid'5000 developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

References

1. Aubanel, E., Wu, X.: Incorporating latency in heterogeneous graph partitioning. In: IPDPS 2007, pp. 1–8 (2007)
2. Beaumont, O., Boudet, V., Rastello, F., Robert, Y.: Matrix multiplication on heterogeneous platforms. IEEE Trans. Parallel Distrib. Syst. 12(10), 1033–1051 (2001)
3. Catalyurek, U., Boman, E., Devine, K., et al.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In: IPDPS 2007, pp. 1 –11 (2007)
4. Chevalier, C., Pellegrini, F.: PT-Scotch: A tool for efficient parallel graph ordering. Parallel Computing 34(6-8), 318–331 (2008)
5. Choi, J.: A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In: HPC Asia 1997, pp. 224–229 (1997)
6. Clarke, D., Lastovetsky, A., Rychkov, V.: Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms. Parallel Processing Letters 21, 195–217 (2011)
7. Clarke, D., Lastovetsky, A., Rychkov, V.: Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models. In: Alexander, M., et al. (eds.) Euro-Par 2011, Part I. LNCS, vol. 7155, pp. 450–459. Springer, Heidelberg (2012)
8. Fatica, M.: Accelerating Linpack with CUDA on heterogenous clusters. In: GPGPU-2, pp. 46–51. ACM (2009)
9. Karypis, G., Schloegel, K.: ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. Version 4.0 (2013), `http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf`
10. Lastovetsky, A., Reddy, R.: Data partitioning with a functional performance model of heterogeneous processors. Int. J. High Perform. C 21, 76–90 (2007)
11. 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)
12. Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: MICRO-42, pp. 45–55 (2009)
13. Malony, A.D., Biersdorff, S., Shende, S., et al.: Parallel performance measurement of heterogeneous parallel systems with GPUs. In: ICPP 2011, pp. 176–185 (2011)
14. Ogata, Y., Endo, T., Maruyama, N., Matsuoka, S.: An efficient, model-based CPU-GPU heterogeneous FFT library. In: IPDPS 2008, pp. 1 –10 (2008)
15. Rychkov, V., Clarke, D., Lastovetsky, A.: Using multidimensional solvers for optimal data partitioning on dedicated heterogeneous HPC platforms. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 332–346. Springer, Heidelberg (2011)
16. Walshaw, C., Cross, M.: Multilevel mesh partitioning for heterogeneous communication networks. Future Generation Comput. Syst. 17(5), 601–623 (2001)
17. Yang, C., Wang, F., Du, Y., et al.: Adaptive optimization for petascale heterogeneous CPU/GPU computing. In: Cluster 2010, pp. 19–28 (2010)
18. Zhong, Z., Rychkov, V., Lastovetsky, A.: Data partitioning on heterogeneous multicore platforms. In: Cluster 2011, pp. 580–584 (2011)
19. Zhong, Z., Rychkov, V., Lastovetsky, A.: Data partitioning on heterogeneous multicore and multi-GPU systems using functional performance models of data-parallel applications. In: Cluster 2012, pp. 191–199 (2012)

DISBench: Benchmark for Memory Performance Evaluation of Multicore Multiprocessors

Alexander Frolov and Mikhail Gilmendinov

JSC NICEVT, Moscow, Russia
{frolov,gilmendinov}@nicevt.ru

Abstract. In this paper we present DISBench, an open-source benchmark suite designed for memory performance evaluation of multicore processors and multiprocessor systems on different sets of workload. DISBench includes three memory access kernels: stream, stride and random, which represent different types of memory intensive applications. DISBench natively supports hardware performance counters for detailed performance analysis. Evaluation results of the modern multicore processors (Intel Sandy Bridge-EP and AMD Interlagos) are presented and discussed.

Keywords: multiprocessor, multicore, bandwidth, benchmarking.

1 Introduction

For the last few years emergence of data intensive applications for HPC systems has forced the development of new architectures suitable for its requirements as well as testing and evaluation tools.

Most data intensive applications are characterized with a prevalence of memory access instructions with low temporal and space locality, which is a unsuitable way of operating for memory subsystem of current microprocessor architectures optimized for cache-friendly access patterns.

Current microprocessors (Intel and AMD) have hierarchically organized caches (L1, L2 and L3) and integrated memory controller which provides exchange of 64 byte memory blocks (cache lines) with offchip memory.

There are few benchmarks which specified for memory subsystem evaluation of multicore multiprocessor systems. Most of them are specialized for sequential access patterns such as STREAM [1] or LMbench [4] with exception of RandomAccess [3] which implements irregular random access pattern. Together these benchmarks are a powerful tool for investigating capabilities of hardware platforms. However performance analysis with these benchmark can be worksome because of the lack of the means for performance inspection as well as non-sufficient control over memory allocation in NUMA systems and thread affinity.

We developed new open-source benchmark called DISBench specified for memory performance evaluation on diverse access patterns. Currently in DISBench three synthetic patterns are implemented: stream, stride and random.

Also extensive performance analysis is provided through support of hardware performance counters. Currently different level cache miss and TLB miss counters are supported which proved to simplify analysis of the results. We performed memory subsystem performance analysis of the up-to-date multicore microprocessors Intel Sandy Bridge-EP and AMD Interlagos.

This paper makes the following contributions:

– We developed new open-source benchmark DISBench for memory performance analysis with support of hardware performance counters.
– We evaluated Intel Sandy Bridge-EP and AMD Interlagos platforms on DISBench and analysed obtained results. In the scope of this paper we limited evaluation with only local memory.

The rest of the paper is organized as follows. Section 2 presents overview of existing benchmarks oriented on memory performance evaluation. DISBench and its kernels are described in Section 3. Section 4 presents configuration of hardware platforms. Evaluation results and its analysis are presented in Section 5. Section 6 concludes this paper and provides plans for the future work.

2 Related Work

Benchmarking [or microbenchmarking] is a well known and widely adopted technique to understand characteristics of hardware architecture and analyze performance of real life applications. Memory subsystem is a key part of system architecture which is frequently becomes bottleneck for wide class of applications. There are multiple tools and methods to analyze performance of memory subsystem of multicore multiprocessor compute nodes.

One of the best known and widely used memory targeted benchmarks is STREAM [1], [2] which is used to measure bandwidth of memory subsystem under serial access patters. STREAM is a multithreaded program developed on OpenMP. There is no possibilities to control distribution of allocated memory on NUMA nodes in STREAM except with external utilities such as numactl. In most cases STREAM is used to measure performance of local memory in SMP multiprocessors.

Another popular microbenchmark is RandomAccess from HPC Challenge benchmark suite [3]. RandomAccess is used to measure performance of memory subsystem under uniformly randomized accesses pattern with load-modify-store (update) operation. In HPC Challenge serial (single threaded) and MPI parallel implementation of RandomAccess are included.

Benchmark suite LMbench [4] includes microbenchmarks to measure bandwidth and latencies of different system software routines. LMbench measures memory bandwidth for read, write and copy operations as well as memory read latency.

Another interesting benchmark is x86membench [5],[6] from BenchIT benchmark suite which is a set of low level primitives written on assembler code used to measure bandwidth and latencies on different levels of memory hierarchy.

3 DISBench

DISBench is an open-source benchmark written on C++ (all kernels are pure C functions) with command line interface and extensive set of parameters. DISBench is designed for memory performance evaluation of multicore multiprocessor systems (SMP and NUMA compute nodes) with advanced performance analysis through incapsulating of hardware performance counters into kernel codes.

Currently DISBench comprises of the three synthetic kernels: stream, stride and random. All of them can use one of the three memory access operations: load, store or load-modify-store. Each kernel supports multithreading execution implemented by POSIX threads library.

DISBench supports control of core allocation for threads to run: each thread can be bind to specific or automatically scheduled by OS. This allows to omit negative effects of OS scheduling on performance results.

DISBench supports control over sharing of memory by threads: memory can be shared or private. Additionally DISBench supports control over allocation of memory on NUMA nodes by using numalib API which is bit more accurate than numactl utility. In DISBench three modes of memory distribution are implemented: OS driven, blocked and interleaved with specified block size multiple of page size.

DISBench supports hardware performance counters through PAPI interface [7], which enables easier performance analysis. Currently following performance counters are supported: L1 miss counter, L2 miss counter, L3 miss counter and TLB miss counter.

3.1 Stream Kernel

Stream kernel is used to estimate sustained bandwidth of the memory subsystem under access pattern with stream of serial addresses. Stream kernel consists of a simple loop which reads or writes to consecutive array elements (access pattern is similar as in STREAM benchmark). Memory is divided between threads by blocks of equal size, each thread works with its own contiguous part of memory. Manual unrolled versions of stream kernel are also implemented with unroll depth of 4, 8 and 16.

3.2 Stride Kernel

Stride kernel is used to estimate sustained performance of the memory subsystem under access pattern with stream of addresses with constant stride. Stride kernel allows to estimate characteristics of hardware prefetching mechanism, analyse conflicts in memory controller, study TLB miss handling effects and so on. Manual unrolled versions of stream kernel are also implemented with unroll depth of 4, 8 and 16.

Table 1. Configuration of test systems

Vendor	Intel	AMD
Processor	Xeon E5-2660	Opteron 6234
Codename	Sandy Bridge-EP	Interlagos
Sockets	1	4
Cores	8	2x 6
Clock speed	2.2 Ghz	2.4 Ghz
L1 data cache	8x 32 KB (8-way)	12x 16 KB (4-way)
L2 cache	8x 256 KB (8-way)	12x 1MB
L3 cache	20 MB (20-way)	2x 8M
IMC channels	4	2x 2
DIMMs/channel	2x 8GB DDR3-1600	8x 8GB DDR3-1600
Compiler	GCC 4.3.4	GCC 4.3.4
Operating system	SUSE 11 (x86_64, PAE)	SUSE 11 (x86_64, PAE)

3.3 Random Kernel

Random kernel is used to estimate sustained performance of the memory subsystem under access pattern with stream of random addresses. Random addresses are generated as in HPCC RandomAccess. There are two ways to run random kernel: first is to generate indices on-the-fly and second is to use index vector with pregenerated indices.

4 Test Platforms

Table 1 shows hardware configuration of the platforms used for performance evaluation. We used modern multisocket compute nodes with up-to-date microprocessors from Intel and AMD.

5 Evaluation Results

In this section we present the results of the series of measurements which were performed on several multicore multiprocessor platforms. The main purpose of the evaluation was to estimate effectiveness of memory subsystem under various workloads and through obtained results to make assumptions about microarchitecture of modern microprocessor (Intel Sandy Bridge-EP and AMD Interlagos) system and compare it to each other.

In all cases discussed below we used only single processor of multiprocessor node and evaluated its local memory subsystem. Evaluation of memory performance in multiprocessor mode is out of the scope of this paper and will be done in future work.

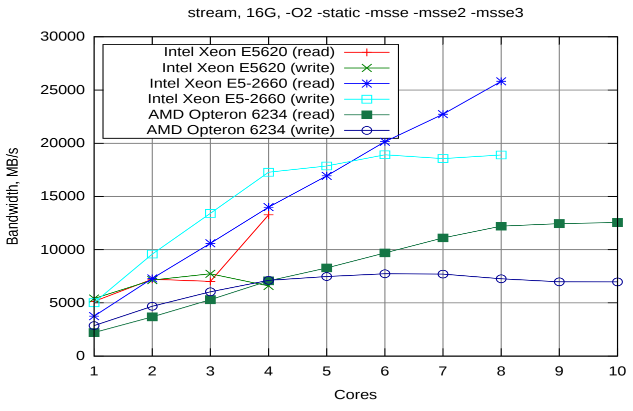


Fig. 1. Performance of stream kernel

5.1 Stream Kernel

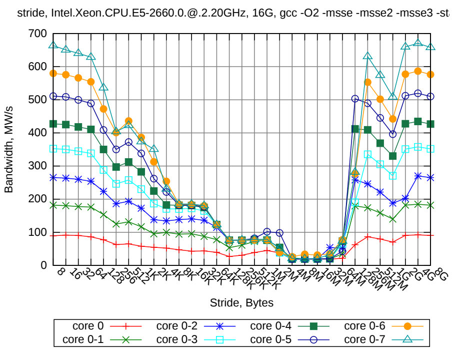
Fig. 1 shows scalability of sustained bandwidth of Sandy Bridge-EP, AMD Interlagos on stream kernel for read and write operations with the number of used micropocessor cores. We used 16 GB of memory for the test and applied manual loop unrolling (16 iterations).

As it can be seen from the Fig. 1 both Sandy Bridge-EP and Interlagos showed similar behaviour. For the small number of cores write operations achieved better sustained bandwidth than read operation and for the high number read operations overcome write operations. It can be explained by two reasons. First is that the number of issued instruction for the write operations is greater than for read operations because of dependencies in loop iterations when read operations are used. That is the reason for higher bandwidth in the case of write operations and small number of cores. Second is that store operation involves double transfer of cache line between memory controller and offchip memory (reading data from memory and writing it back to memory) while load operation involves only single transfer of data that is the reason of higher bandwidth in the case of read operations and high numbers of used cores.

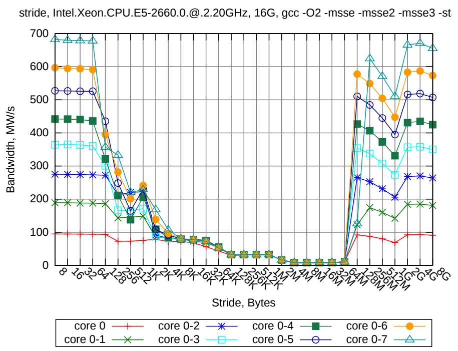
5.2 Stride Kernel

Fig. 2 shows the results of stride kernel obtained on Sandy Bridge-EP and Interlagos processors correspondingly. We used 16 GB of memory for the benchmark and did not apply loop unrolling. For Interlagos both 6-core nodes of the socket have been used and memory was divided between them equally i.e. 8 GB per node. Also we used start offset of 128 bytes for each core.

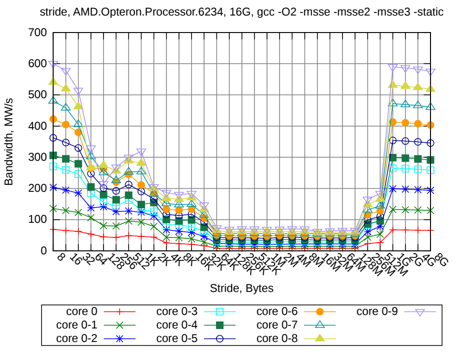
There are a whole set of architectural features such as hardware prefetching mechanism, channel and bank interleaving, TLB structure and page map organization, data cache sizes which influence on the results obtained on this kernel.



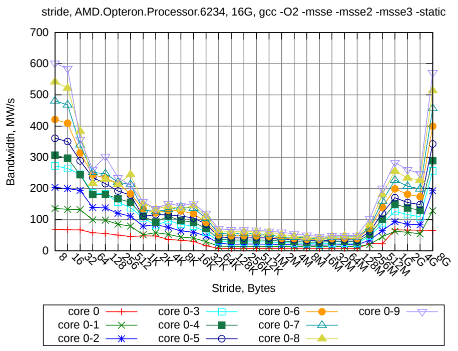
(a) Sandy Bridge-EP, loads



(b) Sandy Bridge-EP, stores



(c) Interlagos, loads

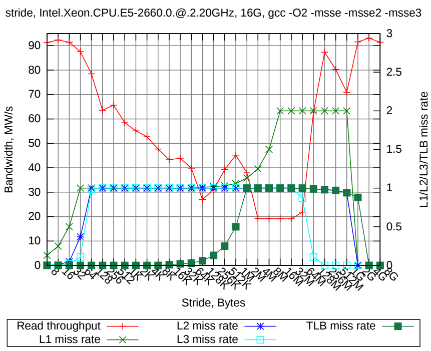


(d) Interlagos, stores

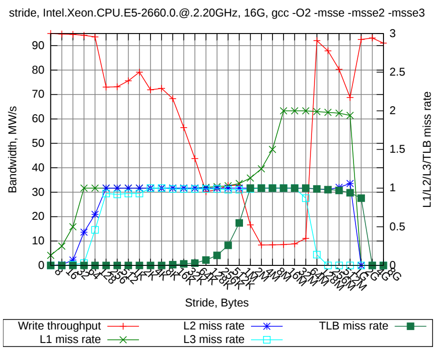
Fig. 2. Sustained bandwidth for stride kernel on Intel Sandy Bridge-EP for loads (a) and stores (b) and AMD Interlagos for loads (c) and stores (d)

As it can be seen from the Fig. 2 all plots have a similar shape which consists of three phases: decrease, minimum and increase. For Sandy Bride-EP (see Fig. 2 (a),(b)) decrease phase is on the interval from 8 to 4M, minimum – from 4M to 64M and increase – from 64M to 8G. For Interlagos (see Fig. 2 (c),(d)) decrease – from 8 to 64K, minimum – from 64K to 128M and increase – from 128M to 8G.

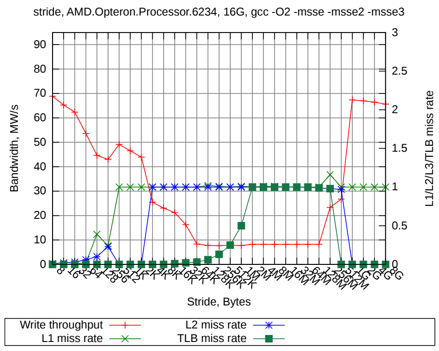
At the decrease phase we see gradual decreasing of sustained bandwidth with increasing of stride length. These can be explained by two factors. First is cache line usage and hardware prefetch efficiency. When stride is very small (8, 16 or 32 bytes) we access cache line [already moved to L1 cache] multiple times (8, 4 or 2 correspondingly) which improves performance. When we increase stride to 64 bytes and more each loaded cache line is used only once. Hardware prefetch efficiency drops down with further increasing of stride length. Second factor is a memory controller channel and memory bank conflicts which increases with



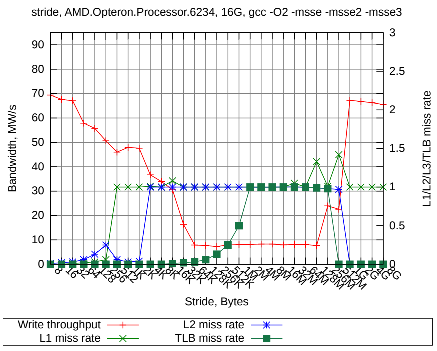
(a) Sandy Bridge-EP, loads



(b) Sandy Bridge-EP, stores



(c) Interlagos, loads



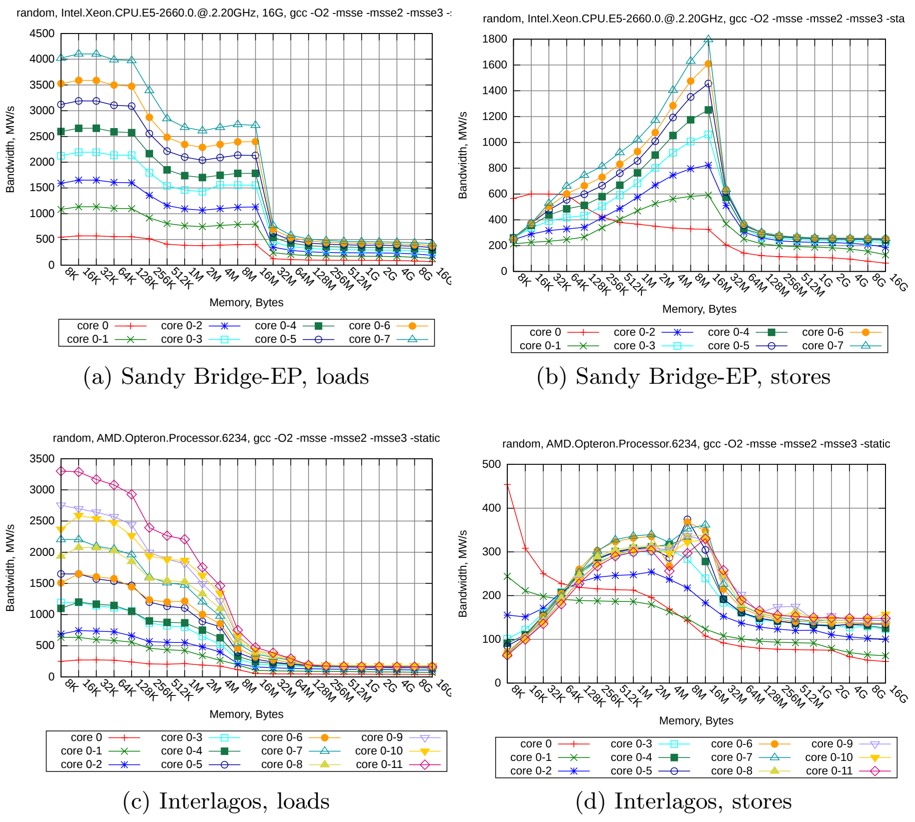
(d) Interlagos, stores

Fig. 3. Sustained bandwidth and performance counters (L1, L2, L3 and TLB miss rates) for stride kernel on Intel Sandy Bridge-EP for loads (a) and stores (b) and AMD Interlagos for loads (c) and stores (d)

increase of stride till all accesses are directed to the single bank and all memory subsystem works with a bandwidth of single bank (on the plots this corresponds to the minimum phase).

At the increase phase we see steep increasing of sustained bandwidth which occurs because of reduction of work set (the greater stride is the lesser memory cells are accessed) and location of all the data in caches. However it worthwhile to note that performance begins to rise only from 64M for Sandy Bridge-EP (see Fig. 2 (a), (b)) and from 128M for Interlagos (see Fig. 2 (c), (d)) when relation of 16G to 128M is 128 which means that only 128 cache line are being accessed. We assume that the reason is the perpetual conflicts in set-associative cache memories i.e. accesses to the limited number of sets because of some bits in index offset preserves its values since stride number is a power of two number.

On the Fig. 3 results of stride kernel with information about L1, L2 and L3 cache and TLB relative miss rates Sandy Bridge-EP and Interlagos are presented. We calculated relative miss rates as a ratio of total misses to total number of



(a) Sandy Bridge-EP, loads (b) Sandy Bridge-EP, stores

(c) Interlagos, loads (d) Interlagos, stores

Fig. 4. Sustained bandwidth for random kernel on Intel Sandy Bridge-EP for loads (a) and stores (b) and AMD Interlagos for loads (c) and stores (d)

memory accesses. In both cases single core has been used. We have not been able to measure number L3 cache misses for Interlagos as it is currently not supported by PAPI library.

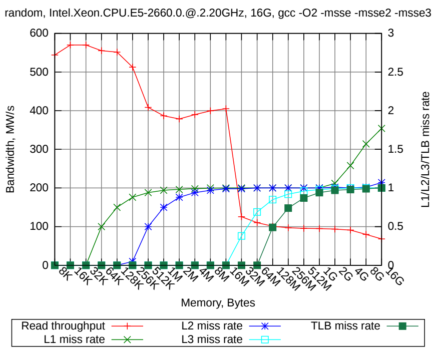
Analysing the behaviour of performance counters we can prove and extend explanation of memory subsystem effects presented above. For example, for Sandy Bridge-EP (see Fig. 3 (a),(b)) L1 miss rate increases from 0 to 1.0 when stride length has been increased to 64 bytes and greater. It means that each memory access involves L1 miss and causes loading of a new cache line from the memory.

At the same time to L2 and L3 miss rates become 1.0 when stride length has been increased to 128 bytes and greater. We assume that this effect caused by adjacent cache line prefetch mechanism which in Sandy Bridge-EP loads additional cache line only to L2 and L3 caches. The value of L3 miss rate drops to 0.0 again when stride is increased to 128M and greater which means that there is no more any misses to L3, as for L1 and L2 miss rates its values are set to 0.0 after stride reaches 2G. Thus assumption about conflicts in caches stated earlier is proved because 16G/2G=8 and L1 and L2 are 8-way associative.

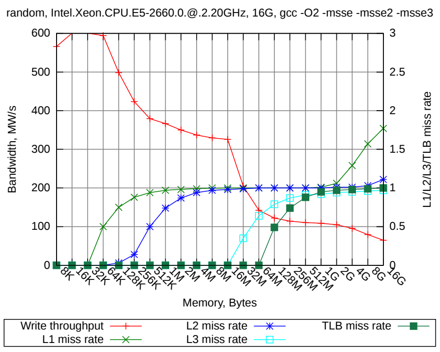
As for TLB miss rate it becomes equal to 1.0 when stride length increases to 2M which means that huge pages are allocated by Linux kernel memory manager automatically. Furthermore when stride length is greater than 16M with every TLB miss there is an additional miss to L1 when page map is accessed to load page descriptor to TLB and L1 miss rate becomes 2.0. This is due to the fact that each cache line holds 8 page descriptors which are 64-bit in PAE mode. When stride is 16M or greater and huge pages are used each TLB miss generate access to new cache line and L1 miss.

5.3 Random Kernel

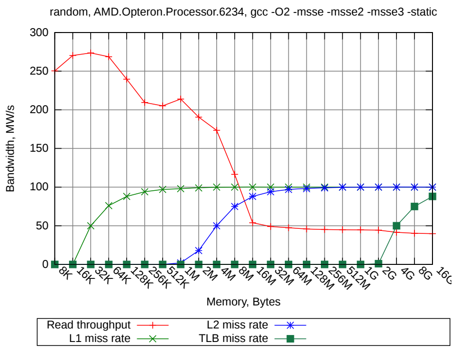
Fig. 4 shows the results of random kernel obtained on Sandy Bridge-EP and Interlagos processors. As in the previous cases we used 16 GB of memory for the



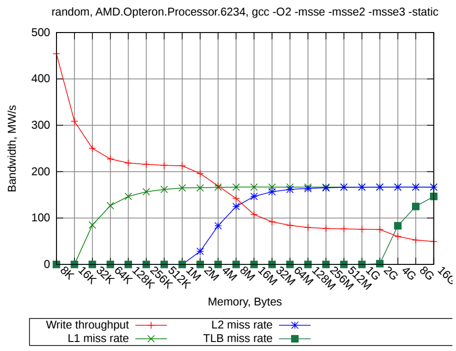
(a) Sandy Bridge-EP, loads



(b) Sandy Bridge-EP, stores



(c) Interlagos, loads



(d) Interlagos, stores

Fig. 5. Sustained bandwidth and performance counters (L1, L2, L3 and TLB miss rates) for random kernel on Intel Sandy Bridge-EP for loads (a) and stores (b) and AMD Interlagos for loads (c) and stores (d)

benchmark. For Interlagos both nodes of the socket have been used. For index generating we used on-the-fly HPCC Challenge random generator during the execution.

For load operations (see Fig. 4 (a) and Fig. 4 (c)) in both cases there can be seen relatively slow degradation of performance with increase of used memory in the range from 8K to 16M when data is fully stored in cache memories. From 32M to 16G there sustained bandwidth drops and shows the performance of memory controller and offchip memory bus. For Sandy Bridge-EP 450 MW/s (3,6 GB/s) was obtained on eight cores, for Interlagos – 220 MW/s (0,9 GB/s) on twelve cores.

For store operations (see Fig. 4 (b) and Fig. 4 (d)) on the range from 8K to 16M performance increases for both Sandy Bridge-EP and Interlagos. This is is related to L1 and L2 cache coherency support as store operations involves invalidation of cache lines in each core and broadcasting of information about of cache line modification to other cores. This greatly reduces performance especially for small memory sizes as every store operation introduces cache-coherence overheads. From 32M to 16G there sustained bandwidth drops to 200 MW/s for Sandy Bridge-EP on eight cores and to 140 MW/s for Interlagos on twelve cores.

On the Fig. 5 results of random kernel with information about L1, L2 and L3 cache and TLB relative miss rates Sandy Bridge-EP and Interlagos are presented. As it can be seen from the figures L1 cache size is 32K per core for Sandy Bridge-EP and 16K per core for Interlagos, L2 – 256K per core for Sandy Bridge-EP and 1M per compute module (two Bulldozer cores) for Interlagos and L3 – 16M for Sandy Bridge-EP (actual size of L3 is 20M which is not power of two and can not be seen on the logarithmic scale).

It can be seen from the for Fig. 5 that TLB reach for Sandy Bridge-EP is 64M and when for Interlagos TLB reach is 2G, however this did not improved overall results for Interlagos comparing to Sandy Bridge-EP.

6 Conclusion

In the paper we presented new open-source benchmark DISBench for memory subsystem performance evaluation of multicore multiprocessor compute nodes. DISBench can be used in the development of data-intensive applications for enhanced performance analysis. DISBench includes three synthetic kernels: stream, stride and random, as well as allows to choose one of the three memory operations (load, store, load-modify-store). DISBench supports hardware performance counters which provide opportunity for deeper inspection of results.

Evaluation of Intel Sandy Bridge-EP and AMD Interlagos with DISBench was performed. Results showed moderate advancement of Sandy Bridge-EP on Interlagos on each of the kernels. However Interlagos showed very large TLB reach and effectiveness of hardware prefetching. Hardware performance counter statistics enabled to simplify result analysis and to make some assumptions about processors microarchitecture.

In the future we plan to use DISBench extensively for multiprocessor evaluation and extend its functionality and usability of DISBench as well as to add new kernels and support of different architectures (such as Nvidia Tesla or Intel MIC coprocessors).

References

1. McCalpin, J.D.: Stream: Sustainable memory bandwidth in high performance computers. University of Virginia, Tech. Rep. (1991-2007), `http://www.cs.virginia.edu/stream/`
2. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Sociaty Technical Committee on Computer Architecture (TCCA), Newsletter, pp. 19–25 (December 1995)
3. Dongarra, J.J., Luszczek, P.: Introduction to the HPC Challenge Benchmark Suite
4. McVoy, L.W., Staelin, C.: lmbench: Portable Tools for Performance Analysis. In: USENIX Annual Technical Conference, pp. 279–294 (1996)
5. Molka, D., Hackenberg, D., Schöne, R., Müller, M.S.: Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, PACT 2009, pp. 261–270. IEEE (2009)
6. Hackenberg, D., Molka, D., Nagel, W.E.: Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In: Proceedings of the 42nd International Symposium on Microarchitecture, MICRO 2009, pp. 413–422. ACM (2009)
7. London, K., Moore, S., Mucci, P., Seymour, K., Luczak, R.: The PAPI Cross-Platform Interface to Hardware Performance Counters. Department of Defense Users' Group Conference Proceedings, Biloxi, Mississippi, June 18-21 (2001)

Fast and Scalable, Lock-Free k-FIFO Queues

Christoph M. Kirsch, Michael Lippautz, and Hannes Payer

University of Salzburg
`firstname.lastname@cs.uni-salzburg.at`

Abstract. We introduce fast and scalable algorithms that implement bounded- and unbounded-size lock-free k-FIFO queues on parallel, shared memory hardware. Logically, a k-FIFO queue can be understood as queue where elements may be dequeued out-of-order up to $k-1$, or as pool where the oldest element is dequeued within at most k dequeue operations. The presented algorithms enable up to k enqueue and k dequeue operations to be performed in parallel. Unlike previous designs, however, the algorithms also implement linearizable emptiness (and full) checks without impairing scalability. We show experimentally that there exist optimal and robust k that result in best access performance and scalability. We then demonstrate that our algorithms outperform and outscale all state-of-the-art concurrent pool and queue algorithms that we considered in all micro- and most macrobenchmarks. Moreover, we demonstrate a prototypical controller which identifies optimal k automatically at runtime achieving better performance than with any statically configured k.

1 Introduction

We are interested in the design and implementation of fast concurrent pools and queues whose access performance scales with the number of available processing units on parallel, shared memory hardware. We introduce two algorithms that implement bounded-size (BS) and unbounded-size (US), lock-free k-FIFO queues with linearizable emptiness (and full) check. The BS algorithm maintains a bounded-size array of elements that is dynamically partitioned into segments of size k called k-segments while the US algorithm maintains an unbounded list of k-segments. Thus up to k enqueue and k dequeue operations may be performed in parallel. The US algorithm simplifies for $k=1$ to an algorithm that implements a lock-free FIFO queue similar to the lock-free Michael-Scott FIFO queue (MS) [1] but without a sentinel node. See Section 3 for more details.

The idea of k-segments has first appeared in the so-called Segment Queue (SQ) [2]. The US algorithm uses k-segments but improves upon SQ in performance and scalability through less overhead and reduced contention by performing removal of elements from k-segment slots lazily. Dequeue operations only perform costly compare-and-swap operations when used k-segments slots are actually found. The enhancement is necessary to obtain positive scalability on the hardware and for the workloads that we considered. Moreover, both k-FIFO queue algorithms implement a linearizable emptiness check which SQ does not. In other words, upon dequeueing attempts SQ may not always find and retrieve elements but instead return nothing even though the queue is not empty. A linearizable emptiness check may be required for implementing application requirements such as termination. We discuss the relation to SQ in detail in Section 4.

In Section 5, before presenting a detailed performance analysis of our algorithms relative to a variety of concurrent pool and queue algorithms, we show experimentally that there exist optimal and robust k that result in best performance and scalability. Interestingly, performance generally increases with k but only up to a certain point which is determined by a tradeoff between degree of parallelism and management overhead. Our algorithms outperform and outscale all other algorithms that we considered in almost all threading and contention scenarios. Finally, we discuss a prototypical controller that adjusts k dynamically and automatically at runtime outperforming any static and manual k configuration on the workload that we considered.

2 k-FIFO Queue Sequential Specification

A k-FIFO queue is a restricted out-of-order k-relaxation of a FIFO queue as defined in the framework for quantitative relaxation of concurrent data structures [3]. We informally discuss the sequential specification of a k-FIFO queue.

Let Σ bet the set of queue methods with input and output values defined as

$$\Sigma = \{\texttt{enq(x)}, \texttt{deq(x)} \mid x \in D\} \cup \{\texttt{deq(null)}\}$$

where D is the set of elements that can be enqueued and dequeued from the queue and `deq(null)` represents a dequeue returning empty. We refer to sequences in Σ^* as queue sequences.

The sequential specification of a FIFO queue is the set $S \subseteq \Sigma^*$ that contains all valid FIFO queue sequences. Informally, valid FIFO queue sequences are sequences where elements are enqueued in the same order as they are dequeued. Furthermore, a `deq(null)` only happens if the queue is empty at the time of `deq(null)`, i.e., every element which gets enqueued before `deq(null)` also gets dequeued before `deq(null)`. For example, the queue sequence

$$\texttt{enq(a)enq(b)deq(a)enq(c)deq(b)deq(c)}$$

belongs to S whereas

$$\texttt{enq(a)enq(b)deq(b)enq(c)deq(a)deq(c)}$$

does not.

A restricted out-of-order k-relaxation of a FIFO queue is the set $S_k \subseteq \Sigma^*$ containing all sequences with a distance of at most k to the sequential specification S of the queue. Informally, the distance is the number of elements overtaking each other in the queue, i.e., an element e may overtake at most $k-1$ elements and may be overtaken by at most $k-1$ other elements before it is dequeued. A 1-FIFO queue thus corresponds to a regular FIFO queue. The above example sequences are therefore within the specifications of a 1-FIFO and a 2-FIFO queue, respectively. We show in Section 3.2 that our k-FIFO queue algorithms indeed implement k-FIFO queues.

3 k-FIFO Queue Algorithms

We present the algorithms of the lock-free bounded-size (BS) and unbounded-size (US) k-FIFO queues for $k > 0$. The pseudo code of the algorithms is depicted in Listing 1.1. The occurrence of the ABA problem is made unlikely through version numbers (also known as ABA counters). We refer to values enhanced with version numbers as atomic values. We use compare-and-swap (CAS) operations to atomically swap in values at given locations. The gray highlighted code is only used in the BS version. We present the general idea of the algorithms followed by a detailed discussion of the BS algorithm. We then discuss the US algorithm by outlining its differences to the BS algorithm and finally show informally that both algorithms are linearizable with respect to the k-FIFO specification.

FIFO queues are usually implemented using head and tail pointers, where elements are dequeued at the head and enqueued at the tail pointer. When implementing a linearizable FIFO queue, these head and tail pointers may become scalablity bottlenecks. The principle idea of the BS and US k-FIFO queue algorithms is to reduce contention on the head and tail pointers by maintaining an array (BS) or a list (US) of so-called k-segments each consisting of k slots, instead of maintaining an array or list of plain queue elements. A slot may either point to `null` indicating an empty slot or may hold a so-called item, which is our implementation concept for enqueued elements. An enqueue operation is served by the tail k-segment and a dequeue operation is served by the head k-segment. Hence, up to k enqueue and k dequeue operations may be performed in parallel.

The BS k-FIFO queue algorithm is based on an array of atomic values of a given `size`. For simplicity we restrict `size` to be a multiple of k. The queue `tail` and queue `head` pointers are also atomic values. Both initially point to the slot at index zero.

The `enqueue` method is depicted in Listing 1.1. Given an `item` representing an element to be enqueued, the method returns `true` when the `item` is successfully inserted and `false` when the queue is full. First the method tries to find an empty slot in the tail k-segment which is located in between the indices $[\texttt{tail}, \texttt{tail} + k[$ using the `find_empty_slot` method (line 5). The `find_empty_slot` method randomly selects an index in between $[\texttt{tail}, \texttt{tail} + k[$ and then linearly searches for an empty slot starting with the selected index wrapping around at index $\texttt{tail} + k - 1$. Afterwards the `enqueue` method checks if the k-FIFO queue state has been consistently observed by checking whether `tail` changed in the meantime (line 6) which would trigger a retry. If an empty slot is found (line 7) the method tries to insert the `item` at the location of the empty slot using a `CAS` operation (line 9). If the insertion is successful the method verifies whether the insertion is also valid by calling the `committed` method (line 10), as discussed below. The `enqueue` method retries in the following scenarios. If no empty slot is found in the current tail k-segment the `enqueue` method tries to increment `tail` by k using `CAS` (line 19) and then retries. If `tail` cannot be incremented without overtaking `head` (line 13) and the k-segment to which `head_old` points is empty (line 14) the method tries to increment `head` by k using `CAS` (line 18). If this k-segment is not empty and `head_old` did not change in the meantime (line 15) the queue is full and `false` is returned (line 16).

The committed method (line 21) validates an insertion. It returns true when the insertion is valid and false when it is not valid. An insertion is valid if the inserted item already got dequeued at validation time by a concurrent operation (line 22, 30, 36) or the tail k-segment where the item was inserted is in between the current head k-segment and the current tail k-segment but not equal to the current head k-segment (line 27). If the tail k-segment where the item was inserted is not in between the current head k-segment and the current tail k-segment (line 29) the method tries to undo the insertion using CAS (line 30). If the tail k-segment where the item was inserted is equal to the current head k-segment a race with concurrent dequeueing threads may occur which may not have observed the insertion and may try to advance the head pointer in the meantime. This would result in loss of the inserted item. To prevent that the method tries to increment the version number in the head atomic value using CAS (line 34). If this fails a concurrent dequeue operation may have changed head which would make the insertion potentially invalid. Hence after that the method tries to undo the insertion using CAS (line 36). The committed method returns false (line 38) if the insertion was undone in any of these cases.

The dequeue method is depicted in Listing 1.1. It returns an item if the queue is not empty, and returns null if the queue is empty. Similarly to the enqueue method the dequeue method first tries to find an item in between the indices $[\mathtt{head}, \mathtt{head}+k[$ using the find_item method (line 43). The find_item method randomly selects an index in between $[\mathtt{head}, \mathtt{head}+k[$ and then linearly searches for an item starting with the selected index wrapping around at index $\mathtt{head}+k-1$. Afterwards the dequeue method checks if the queue state has been consistently observed by checking whether head changed in the meantime (line 45) which would trigger a retry. If an item was found (line 46) the method first checks whether head equals tail (line 47), checking whether the queue only consists of a single k-segment. If this is the case the method tries to increment tail by k to prevent starvation of items in the queue and to provide a linearizable emptiness check. Afterwards the method tries to remove the item using CAS (line 50) and returns it if the removal was successful (line 51). If the CAS fails due to a concurrent dequeue, a retry is performed. If no item is found, head equals tail, and tail did not change in the meantime null is returned indicating an empty queue (line 53). This is enough to show that the queue has been empty at the linearization point (see correctness part below). If tail did change in the meantime, the operation tries to increment head by k (line 55) and retries, since no other operation could have enqueued an item into this k-segment anymore.

Note that to hold n items the BS k-FIFO queue has to consist of at least $\lceil \frac{n}{k} \rceil \times k + k$ atomic values. The additional segment, introduced by dequeue (line 47, 48), is necessary to avoid starvation of items which is possible if enqueue and dequeue operations work on the same segment.

3.1 US k-FIFO Queue Algorithm

The US k-FIFO queue algorithm differs from the BS version in the implementation of the committed, advance_tail, and advance_head methods. The gray highlighted code in Listing 1.1 is not used in the US version since there is no full state. Hence the enqueue method always returns true.

Listing 1.1. Lock-free *k*-FIFO queue algorithms

```
enqueue(item):
  while true:
    tail_old = get_tail();
    head_old = get_head();
    item_old, index = find_empty_slot(tail_old, k);
    if tail_old == get_tail():
      if item_old.value == EMPTY:
        item_new = atomic_value(item, item_old.version + 1);
        if CAS(&tail_old->segment[index], item_old, item_new):
          if committed(tail_old, item_new, index):
            return true;
      else:
        if tail_old.value + k == head_old.value:
          if segment_not_empty(head_old, k):
            if head_old == get_head():
              return false;
          else:
            advance_head(head_old, k);
        advance_tail(tail_old, k);

bool committed(tail_old, item_new, index):
  if tail_old->segment[index] != item_new:
    return true;
  head_current = get_head();
  tail_current = get_tail();
  item_empty = atomic_value(EMPTY, item_new.version + 1);
  if in_queue_after_head(tail_old, tail_current, head_current):
    return true;
  else if not_in_queue(tail_old, tail_current, head_current):
    if !CAS(&tail_old->segment[index], item_new, item_empty):
      return true;
  else: //in queue at head
    head_new = atomic_value(head_current.value, head_current.version + 1);
    if CAS(&head, head_current, head_new):
      return true;
    if !CAS(&tail_old->segment[index], item_new, item_empty):
      return true;
  return false;

item dequeue():
  while true:
    head_old = get_head();
    item_old, index = find_item(head_old, k);
    tail_old = get_tail();
    if head_old == get_head():
      if item_old.value != EMPTY:
        if head_old.value == tail_old.value:
          advance_tail(tail_old, k);
        item_empty = atomic_value(EMPTY, item_old.version + 1);
        if CAS(&head_old[index], item_old, item_empty):
          return item_old.value;
      else:
        if head_old.value == tail_old.value && tail_old.value == get_tail():
          return null;
        advance_head(head_old, k);
```

An enqueue operation is served by the tail *k*-segment. When this *k*-segment is full, a new *k*-segment is added to the `tail`. A dequeue operation is served by the head *k*-segment. When this *k*-segment is empty it is removed from the *k*-segment queue except if it is the only *k*-segment in the queue.

Any lock-free FIFO queue algorithm may be used to implement the queue of k-segments. We developed a lock-free FIFO queue which performs better than the lock-free Michael-Scott FIFO queue (MS) [1] when used as k-segment queue in the US k-FIFO queue algorithm. Our implemented k-segment queue always contains at least one usable k-segment, even if this k-segment is empty, enabling fast and direct access to the head and tail k-segments [4].

The US algorithm simplifies for $k = 1$ to an algorithm that implements a lock-free FIFO queue similar to the MS algorithm but without a sentinel node. In contrast to MS, the empty US 1-FIFO queue contains an empty 1-segment in which the first enqueued element is stored. Subsequent dequeue operations may lead to a queue with an empty 1-segment at the head but only because 1-segments are removed lazily (which may also be done eagerly avoiding empty 1-segments altogether in a non-empty queue).

3.2 Correctness

Proposition 1. *The k-FIFO queue algorithms are linearizable with respect to the sequential specification of a k-FIFO queue.*

Proof. Without loss of generality, we assume that each item enqueued in and dequeued from the queue is unique. Furthermore we call a segment s' reachable from a k-segment s, if either $s' = s$, or s' is reachable from $\texttt{s} \rightarrow \texttt{next}$ (US), or s is within the range $[\texttt{head}, \texttt{tail}]$ (BS). Similar, a k-segment s is removed from the queue if it is not reachable from head (US), or not within the range of k-segments $[\texttt{head}, \texttt{tail}]$ anymore (BS).

An item is logically contained in the queue, if `enqueue(i)` has already `committed` and there exists a reachable k-segment containing a segment whose value is `i`. Note that only having a slot containing the item is not enough to guarantee that the item is logically in the queue, because the slot could be in a k-segment that is not reachable anymore (because it has been removed).

We identify linearization points [5] of the `enqueue` and `dequeue` methods depicted in Listing 1.1 to show that the sequential history obtained from a concurrent history by ordering methods according to their linearization points is in the sequential specification of a k-FIFO queue. The linearization point of `enqueue` that inserts an item is the successfully executed `CAS` inserting the item (line 9) if the following call to `committed` validates it and therefore returns `true`. Additionally, the queue may also be full in the BS version. The linearization point of the full check is the last read of an index in `find_empty_slot` (line 5), if there is still an item left in the observed head k-segment (line 14), and the `head` pointer did not change in the meantime (line 15). The linearization point of the `dequeue` method that returns an item is the successfully executed `CAS` with an `EMPTY` item (line 50). The linearization point of the emptiness check is the first read of an index in `find_item` (line 43) in the head k-segment, if the `head` pointer then points to `tail`, and `tail` did not change in the meantime (line 53).

The correctness argument is based on the following facts.

1. An item is enqueued in the queue exactly once. This is a consequence of our unique-items assumption and the control flow of the `enqueue` method that can only modify one slot a time. If the `committed` method identifies an invalid insertion, it removes the item using `CAS` and retries. Hence, at every point in time the item is at most in one k-segment.

2. If an enqueue operation returns full, then during its execution there must be a state at which at least n items have been inserted successfully into the queue (BS only). Since returning full is without any side effect, it suffices to prove the existence of a state which corresponds to a logically full queue. Inserting n items in the queue results in $\lceil \frac{n}{k} \rceil + 1$ used k-segments of which all but the `head` segment contain k items, i.e., they are full. The current `tail` points to the last k-segment before `head` and has been found full. If then the `head` segment also contains at least one item, it is not possible to advance `tail`. Hence, the queue is full.

3. An item is dequeued at most once. If an item `i` is in the queue it can only be removed once, because of 1. and a statement which replaces `i` with `EMPTY`. If `i` is in some slot, but not logically in the queue, then `enqueue` removes it and retries the insertion before committing again. We show that while `i` is in some slot, but not logically in the queue, no `dequeue` can return `i`. Clearly, the call to `committed` has to return `false`, implying that no other thread modifies the slot of `i`. Otherwise, either the first `if` statement (line 22) or the following failed `CAS` attempts (lines 30 and 36) of replacing `i` with `EMPTY` lead to returning `true`. When the control flow reaches the only point for returning `false` in `committed` it is guaranteed that there is no slot containing `i` anymore, making it impossible to dequeue the item.

4. If a dequeue operation returns empty, then during its execution, there must be a state at which there a no items in the queue. Similar to 2. returning empty is without any side effects, so it suffices to show that there exists a state which corresponds to a logical empty state. The queue is empty at the time it observes the first slot in `find_item` as empty, if then all slots are observed as empty and if then the observed `head _old` points to `tail_old` and the actual `tail` did not change in the meantime. This is enough because it guarantees that no slot gets modified by enqueue or dequeue operations until the slots' state is checked.

5. An item j cannot be dequeued before an item i, if they are both in the queue and i, j are in segments s, s′, respectively, with s′ reachable from s and $s \neq s'$. The segment s' can become the head segment only after the segment s has been removed. Moreover, unreachable segments can not contain items that are logically in the queue as validated in `committed`. These observations imply that with respect to linearization points `dequeue(i)` precedes `dequeue(j)`, which only happens if s' is the head segment.

6. An item i can overtake at most $k-1$ other elements. Assume the item i resides in some segment s' reachable from the current head segment s. Because of 5. i cannot overtake any items until s' becomes the head k-segment. In the case that s' becomes the head k-segment, the item i may be dequeued first, effectively overtaking at most $k-1$ other items.

7. An item i is overtaken by at most $k-1$ other items. Again, because of 5. an item i can only be overtaken by elements residing in the same k-segment. As a result i can only be overtaken by at most all $k-1$ other items in the segment. □

The following fairness property follows from Facts 1. and 5.:

Corollary 1. *Dequeuing an element e from a k-FIFO queue may take at most $n+k$ dequeue operations, where n is the number of elements in the queue when e was inserted into the queue.*

Proposition 2. *The k-FIFO queue algorithms are lock-free.*

Similar to others (cf. [1]), one can show that both algorithms are lock-free by demonstrating that whenever some operation retries another operation is making progress. The somewhat lengthy argument can be found elsewhere [4] and is omitted due to space limitations. Note that the US k-FIFO queue allocates memory dynamically through a lock-free allocator.

4 Related Work

We relate our BS and US k-FIFO queue algorithms to existing concurrent data structure algorithms, which we also implemented for and evaluated in a number of experiments in Section 5.

The following queues implement regular unbounded-size FIFO queues: a standard lock-based FIFO queue (LB), the lock-free Michael-Scott FIFO queue (MS) [1], and the flat-combining FIFO queue (FC) [6] (FC). LB locks a mutex for each data structure operation. With MS each thread uses at least two CAS operations to insert an element into the queue and at least one CAS operation to remove an element from the queue. FC is based on the idea that a single thread performs the queue operations of multiple threads by locking the whole queue, collecting pending queue operations, and applying them to the queue. The lock-free bounded-size FIFO queue (BS) [7] is based on an array of fixed size where elements get inserted and removed circularly and enqueue operations may fail when the queue is full, i.e. every array slot holds an element.

The Random Dequeue Queue (RD) [2] is based on MS where the dequeue operation was modified in a way that, given a configurable integer r, a random number in $[0, r-1]$ determines which element is returned starting from the oldest element.

The Segment Queue (SQ) [2] is closely related to the US k-FIFO queue. SQ is based on a list of segments of size $s > 0$ where s is a configurable integer. With SQ an enqueue operation inserts an element at an arbitrary empty position of the youngest segment using a CAS operation. This is similar to our proposed US k-FIFO queue. Analogously, with SQ a dequeue operation logically removes an element at an arbitrary position of the oldest segment using a CAS on a deleted flag. SQ considers all slots for deletion, i.e. it eagerly tries to remove an item (even it already has been removed). T This is different to the US k-FIFO queue where elements are removed lazily, i.e., a CAS operation is only performed on a slot still holding an item.

Experiments reported elsewhere [2] show that in certain configurations (parameters r and s, respectively), RD and SQ scale better than MS. However, the improved scalability comes at the expense of peak performance, as the data also shows that the best overall throughput is reached by MS with only few threads. Neither SQ nor RD reach that peak performance with any number of threads.

In contrast to the US k-FIFO queue, SQ does not provide a linearizable emptiness check. Consider the following example of using SQ: Starting with an empty queue, thread A enqueues the first element into the queue at position $s-1$ of a new segment of size s. After that thread A attempts to perform a dequeue operation by iterating over the slots of the new segment just finding empty slots until slot $s-1$. Right before checking the slot at position $s-1$ thread A gets descheduled. Thread B now performs $s-1$ enqueue operations and fills up the whole segment with elements. After that Thread B

performs a dequeue operation and removes the element from the slot at position $s-1$. Now, thread A wakes up, checks the slot at position $s-1$, encounters that also this slot is empty, and returns *null*. However, at any point in time there was at least one element in the queue. In other words, SQ reported empty although the queue was never empty.

The lock-free linearizable pool (BAG) [8] is based on thread-local lists of blocks of elements. Each block is capable of storing up to a constant number of elements. A thread performing an enqueue operation always inserts elements into the first block of its thread-local list. Once the block is full, a new block is inserted at the head of the list. A thread performing a dequeue operation always tries first to find an element in the blocks of its thread-local list. If the thread-local list is empty, work stealing from other threads' lists is used to find an element. The work-stealing algorithm implements a linearizable emptiness check by repeatedly scanning all threads' lists for elements and marking already scanned blocks which are unmarked when elements are inserted. The implementation works only for a fixed number of threads.

The lock-free elimination-diffraction pool (ED) [9] uses FIFO queues to store elements. Access to these queues is balanced using elimination arrays and a diffraction tree. While the diffraction tree acts as a distributed counter balancing access to the queues, elimination arrays in each counting node increase disjoint-access parallelism. Operations hitting the same index in an elimination array can either directly exchange their data (enqueue meets dequeue), or avoid hitting the counter in the node that contains the array (enqueue meets enqueue or dequeue meets dequeue). If based on non-blocking FIFO queues, the presented algorithm lacks a linearizable emptiness check. If based on blocking queues, there is no empty state at all. Parameters, i.e., elimination waiting time, retries, array size, tree depth, number of queues, queue polling time, need to be configured to adjust ED to different workloads.

The synchronous rendezvousing pool (RP) [10] implements a single elimination array using a ring buffer. Both enqueue and dequeue operations are synchronous. A dequeue operation marks a slot identified by its thread id and waits for an enqueue operation to insert an element. An enqueue operation traverses the ring buffer to find a waiting dequeue operation. As soon as it finds a dequeue operation they exchange values and return. There exist adaptive and non-adaptive versions of the pool where the ring buffer size is adapted to the workload.

5 Experiments

We evaluate the performance and scalability of the BS and US k-FIFO queue algorithms. All experiments ran on an Intel-based server machine with four 10-core 2.0GHz Intel Xeon processors (40 cores, 2 hyperthreads per core), 24MB shared L3-cache, and 128GB of memory running Linux 2.6.39. We implemented a framework to benchmark and analyze different queue and pool implementations under configurable scenarios and workloads. For microbenchmarking the framework emulates a multi-threaded producer-consumer workload where each thread is either a producer or a consumer. The workload can be configured for a different number of threads (n), number of enqueue or dequeue operations each thread performs (o), the computational load performed between each operation (c), the number of pre-filled items (i), and the queue implementation to use.

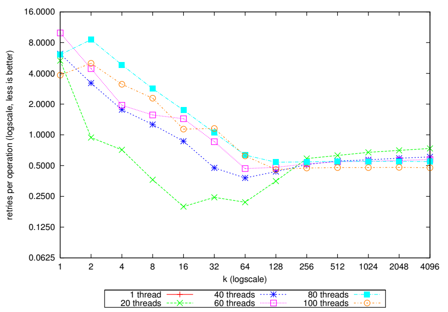
The computational load c between two consecutive operations is created by iteratively calculating π. A computation with $c = 1000$ takes a total of $2300ns$ on average. We fix the operations per thread to $o = 1000000$ and the number of producers and consumers to $\frac{n}{2}$ for all benchmarks. We evaluate the performance and scalability of the queues under low ($c = 10000$), medium ($c = 7000$), high ($c = 4000$) and very high ($c = 1000$) contention. In order to analyze the performance impact and applicability of our k-FIFO queues on real applications we use three different macrobenchmarks: Mandelbrot set calculation, graph traversal computing the spanning tree, and graph traversal computing the transitive closure of a graph.

To avoid paging and caching issues we use our own lock-free allocator, which touches memory pages upon program initialization and can be used to return cache- and page-aligned memory. For the purpose of benchmarking we do not free allocated memory. Freeing memory in lock-free algorithms is an orthogonal problem and can be solved by using hazard pointers [11].

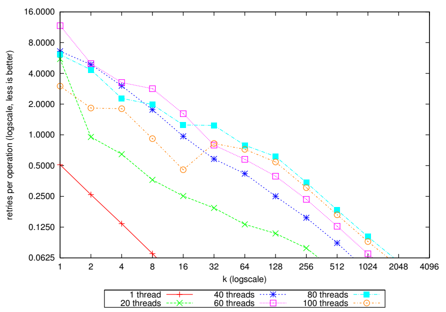
5.1 Understanding *k*

In order to provide a better understanding of the effect of k on performance and scalability we first evaluate the performance of the BS version with increasing k. We omit the measurements for the US version as the results are similar. Performance is measured under very high ($c = 1000$) contention without ($i = 0$) and with ($i = 5000$) pre-filling the queue. Other contention scenarios lead to similar results. We relate the performance of our producer-consumer benchmark, measured in operations per millisecond, to the number of retries per operation and the number of so-called failed reads. Retries are an indicator of contention among `CAS` operations. They occur whenever an enqueue (line 2) or a dequeue (line 39) operation has to take another iteration of the while loop. Failed reads are attempts to find empty slots or items in `find_empty_slot` (line 5) and `find_item` (line 42), respectively. Both retries and failed reads produce overhead and should thus be minimized in order to improve overall performance.

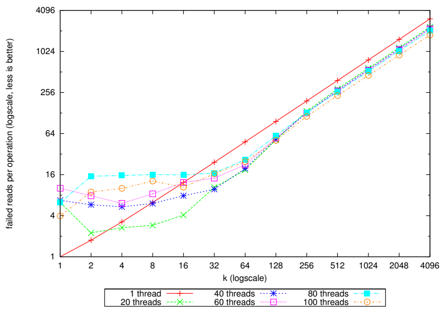
Figure 1 depicts this performance analysis with non-pre-filled results on the left and pre-filled results on the right side. Intuitively, one would expect that a larger k results in fewer retries because of reduced contention among the inserting (line 9) and removing (line 48) `CAS` operations. Figure 1(b) shows that this is true for a setting where the queue is pre-filled with items, i.e., a queue with an initially dense population in the k-segment that is used for dequeueing. However, for a workload where the queue is initially empty there exists a turning point from which the number of retries starts to grow with increasing k. Figure 1(a) depicts this behavior which appears when the k-segment used for dequeueing is only sparsely populated most of the time. In this case the dequeueing operations are likely to contend on the same, rare items in the head k-segment. Figure 1(c) and 1(d) illustrate the number of failed reads. As long as the number of retries is decreasing, failed reads are slowly increasing with larger k since the k-segments to search for items or empty slots get bigger. As soon as the number of retries reaches the turning point in the pre-filled case failed reads are increasing exponentially. Figure 1(e) and 1(f) then visualize the impact of an increasing k on the performance and show that there exists an optimal k with respect to performance. The optimal k is also robust in the sense that there exists only a single range of



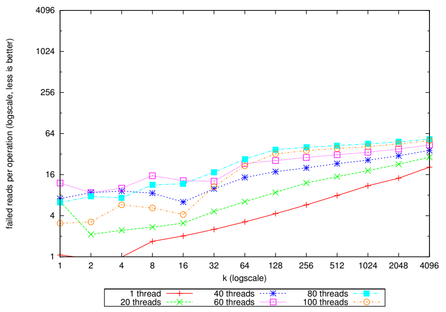
(a) BS number of retries per operation for very high contention ($c = 1000$, $i = 0$)



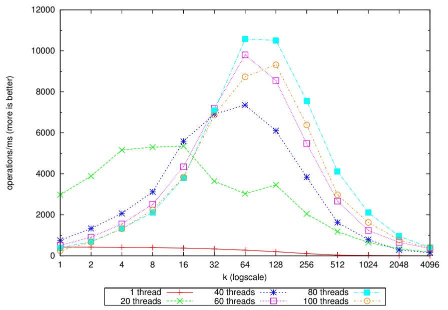
(b) BS number of retries per operation for very high contention ($c = 1000$, $i = 5000$)



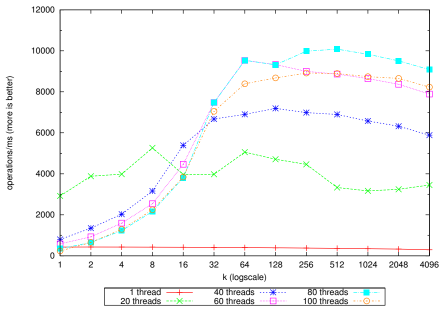
(c) BS number of failed reads per operation for very high contention ($c = 1000$, $i = 0$)



(d) BS number of failed reads per operation for very high contention ($c = 1000$, $i = 5000$)



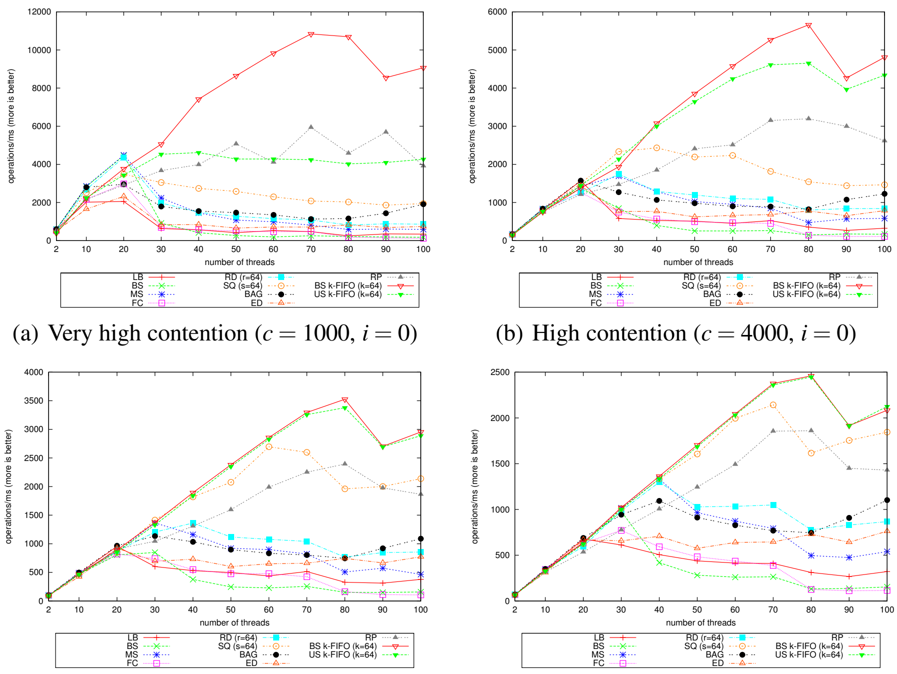
(e) BS performance of very high contention ($c = 1000$, $i = 0$)



(f) BS performance of very high contention ($c = 1000$, $i = 5000$)

Fig. 1. Very high and low contention producer-consumer microbenchmarks with an increasing number of k for different amounts of threads

close-to-optimal k. Furthermore, the population density of a k-segment that is used for dequeueing has an impact on the range of k where good performance can be observed. If k gets too large, i.e., the population in the dequeueing segment is sparse, the performance significantly decreases. The depicted behavior of k suggests a controller that optimizes k to dynamic workloads.



(a) Very high contention ($c = 1000$, $i = 0$)

(b) High contention ($c = 4000$, $i = 0$)

(c) Medium contention ($c = 7000$, $i = 0$)

(d) Low contention ($c = 10000$, $i = 0$)

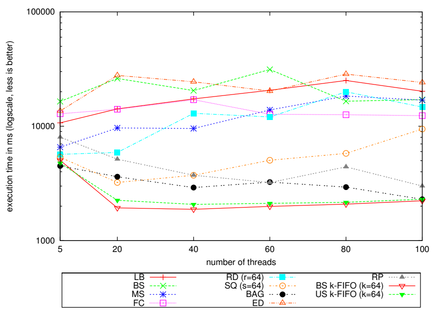
Fig. 2. Performance and scalablity of producer/consumer microbenchmarks with an increasing number of threads

5.2 Performance and Scalability

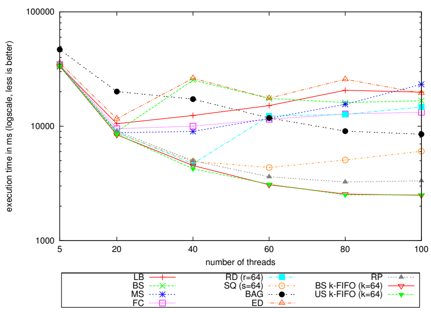
We study the performance of LB, BS, MS, FC, RD, SQ, BS k-FIFO and US k-FIFO queues, and ED, BAG, and RP pools. For the RD, SQ, BS k-FIFO, and US k-FIFO queues we configure $r = s = k = 64$ (see Section 4), which we determined to be a fair and representative configuration for a broad range of thread combinations and workloads. We use the non-adaptive version of the RP algorithm since the number of threads is constant in each run.

Producer-Consumer. Figure 2(a) illustrates the results for the very high contention workload where MS and RD perform best for up to 20 threads. With more than 20 threads scalability is negative for all data structures except RP, BS k-FIFO, and US k-FIFO. The BS k-FIFO queue algorithm is the only algorithm that scales near-linearly.

Similarly, the results with our high contention scenario, depicted in Figure 2(b), show that the scalability turnaround is at 30 threads and that both k-FIFO versions outperform and outscale all other algorithms. As the contention gets less in Figures 2(c) and 2(d), the turnaround gets shifted to a larger number of threads. The difference in performance and scalability of all algorithms is less significant with more computational load. Note that SQ returns up to 2000 times falsely `null` due to the non-linearizable emptiness check.



(a) Low computational load (high contention)



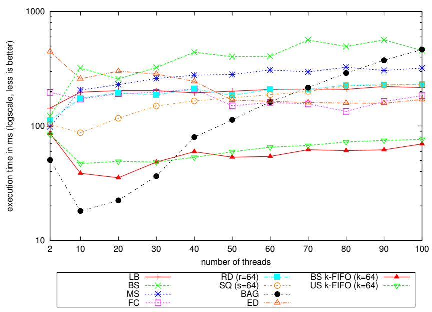
(b) High computational load (low contention)

Fig. 3. Parallel Mandelbrot set calculation with an increasing number of threads (producer-consumer ratio 1:4)

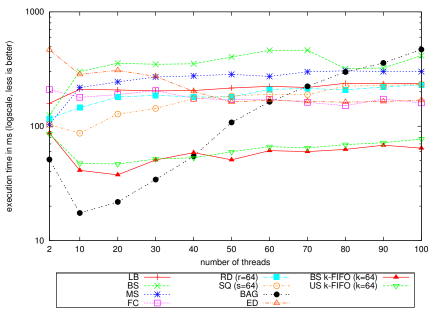
Mandelbrot. We computed and rendered two images of the Mandelbrot set [12] using producer and consumer threads and a shared data structure to distribute the computation across multiple cores. The producer threads divide the image into smaller blocks (4*x*4 pixels in our experiments), write block coordinates in descriptor blocks, and enqueue the descriptor blocks in the shared data structure. The consumer threads dequeue the descriptor blocks from the shared data structure, perform the Mandelbrot calculation on the blocks, and store the results in the corresponding blocks of the final Mandelbrot image. Hence, the workload between the consumer threads is balanced. We use a producer-consumer ratio of 1 : 4 in our experiments, i.e. for each producer thread we add four consumer threads.

The Mandelbrot macrobenchmark results are presented in Figure 3. Each run was repeated 10 times. We present the average execution time of the 10 runs as our metric of performance, less execution time is better. Figure 3(a) shows the performance of the low computational load Mandelbrot benchmark. Low computational load means that Mandelbrot computations are fast for most of the blocks, i.e. the number of iterations in the Mandelbrot calculations is small or zero, and thus the contention on the shared data structure is high. Both *k*-FIFO algorithms show the best performance, followed by the pools BAG and RP. The result of the high computational load Mandelbrot benchmark is depicted in Figure 3(b). High computational load means that the Mandelbrot computations are computationally intensive for most of the blocks, resulting in less contention on the shared data structure. The figure shows that the BS and US *k*-FIFO queues provide the best performance and scalability.

Graph Algorithms. We ran two macrobenchmarks with parallel versions of transitive closure and spanning tree graph algorithms [13] using random graphs consisting of 100000 vertices where 1000000 unique edges got randomly added to the vertices. Non-connected subgraphs are then connected using single edges. The shared data structure is prefilled with 160 randomly determined vertices. From then on each thread iterates over the neighbors of a given vertex and tries to process them (transitive closure or spanning tree operation). Depending on the algorithm, the check whether a neighboring



(a) Transitive closure



(b) Spanning tree

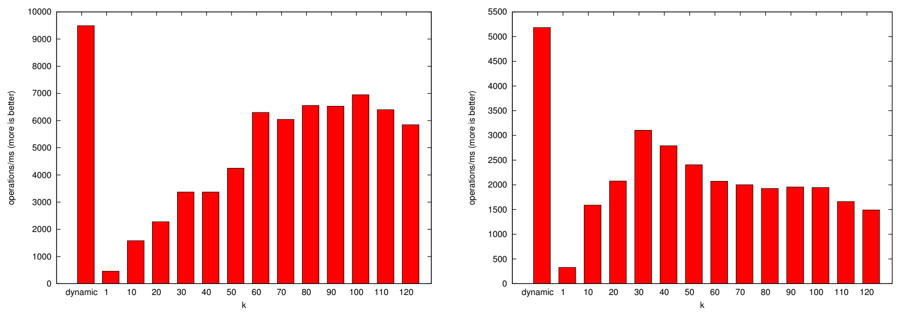
Fig. 4. Performance and scalability of graph traversals on a random graph with 100000 vertices, 1000000 edges, and 160 starting vertices with an increasing number of threads

vertex has already been processed is raceful (transitive closure), or exact (spanning tree). Already processed vertices are then ignored. After processing a vertex, it gets added to the shared data structure. The thread then gets a new vertex from the shared data structure. The graph algorithm terminates as soon as the global queue is empty.

The spanning tree and transitive closure macrobenchmark results are presented in Figure 4. Each run was repeated 10 times. We present the average execution time of the 10 runs as our metric of performance, less execution time is better. The RP pool is not used in this benchmark since it cannot handle a workload where producers are also consumers. Both benchmarks show how the data structures behave under extremely high contention. BAG results in best performance at 10 threads, and then scales negatively. This peak performance is reached because producers are also consumers and thus each thread can access its thread-local list. Note that this result for BAG can only be reached in tailored workloads as the other benchmarks show. In general, all data structures have problems scaling under this high contention. The best performance (except BAG) is reached by the BS and US k-FIFO queues.

5.3 Dynamic k

We implemented a prototypical PID controller which aims at identifying optimal k automatically at runtime for best performance. Each thread i stores performed enqueue operations o_i and performed retries in enqueue operations r_i in thread-local counters. The controller runs in an extra thread, reads the thread-local counters of all n threads periodically (100ms), and resets them to 0 after reading. The goal of the controller is to minimize the ratio $\sum_{i=1}^{n} r_i / \sum_{i=1}^{n} o_i$. The controller operates in the approximately linear part of this ratio. With the US k-FIFO algorithm the controller determines the k-segment size that enqueue operations use to create new segments which store their size for dequeue operations to look up. For the BS k-FIFO algorithm the maximum k needs to be bounded to provide a linearizable emptiness check.



(a) BS k-FIFO queue (b) US k-FIFO queue

Fig. 5. Variable-load producer-consumer microbenchmarks with an increasing number of static k versus a dynamically controlled k

We use a variable-load producer-consumer microbenchmark to evaluate the performance of the controller. Each thread performs $o = 4000000$ operations and starts with $c = 1000$. The workload changes for each thread whenever $o/4$ operations are performed by changing c to 500, 2000, and 1500. We compare the BS and US k-FIFO algorithms with dynamically controlled k to the unmodified baseline versions with statically configured k ranging from 1 to 120. Figure 5(b) shows the performance of the dynamic US k-FIFO queue. Here the controller improves the performance by 60% over the best statically configured configurations. The dynamic BS k-FIFO queue performs about 30% faster than the best statically configured configuration, see Figure 5(a). As explained in Section 5.1 and illustrated in Figure 1, best performance for different levels of contention can be achieved through different configurations of k. The level of contention is proportional to the rate of retries to which k is therefore adjusted dynamically.

6 Conclusions

We have introduced fast and scalable algorithms that implement bounded- and unbounded-size, lock-free, linearizable k-FIFO queues with emptiness (and full) check. We showed experimentally for both algorithms that there exist optimal and robust k that result in best performance and scalability. Moreover, we demonstrated in experiments that our algorithms outperform and outscale many state-of-the-art concurrent queue and pool algorithms on different concurrent producer-consumer workloads. Finding the right k for different workloads is key for best performance and scalability. We suggest to either set k statically to around the number of available parallel processing units or use a controller which automatically adjusts k at runtime as shown in our experiments.

Acknowledgments. This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23). We thank the anonymous referees for their constructive comments and suggestions.

References

1. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. Symposium on Principles of Distributed Computing, PODC, pp. 267–275. ACM (1996)
2. Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: Relaxed consistency for improved concurrency. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 395–410. Springer, Heidelberg (2010)
3. Henzinger, T., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In: Proc. Symposium on Principles of Programming Languages, POPL. ACM (2013)
4. Kirsch, C.M., Lippautz, M., Payer, H.: Fast and scalable k-FIFO queues. Technical Report 2012-04, Department of Computer Sciences, University of Salzburg (June 2012)
5. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst (TOPLAS) 12(3), 463–492 (1990)
6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proc. Symposium on Parallelism in Algorithms and Architectures, SPAA, pp. 355–364. ACM (2010)
7. Colvin, R., Groves, L.: Formal verification of an array-based nonblocking queue. In: Proc. Conference on Engineering of Complex Computer Systems, ICECCS, pp. 507–516. IEEE (2005)
8. Sundell, H., Gidenstam, A., Papatriantafilou, M., Tsigas, P.: A lock-free algorithm for concurrent bags. In: Proc. Symposium on Parallelism in Algorithms and Architectures, SPAA, pp. 335–344. ACM (2011)
9. Afek, Y., Korland, G., Natanzon, M., Shavit, N.: Scalable producer-consumer pools based on elimination-diffraction trees. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part II. LNCS, vol. 6272, pp. 151–162. Springer, Heidelberg (2010)
10. Afek, Y., Hakimi, M., Morrison, A.: Fast and scalable rendezvousing. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 16–31. Springer, Heidelberg (2011)
11. Michael, M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. 15(6), 491–504 (2004)
12. Mandelbrot, B.: Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z. Annals of the New York Academy of Sciences 357, 249–259 (1980)
13. Bader, D., Cong, G.: A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). Journal of Parallel and Distributed Computing 65, 994–1006 (2005)

Development of Basic Approach and Software Package for Effective Parallel Computing of Continuum Mechanics Problems on Hybrid Architecture Systems

Maksim Klimov

Moscow State University, Moscow, Russia

1 Parallel Programming Model

At present the requirements for calculation of continuum mechanics problems are very high: besides the complex geometry and various physical models, actual algorithms on large grids require high performance computing systems. Anyway, due to limited performance of “silicon” core, modern supercomputers represent set of cores and processors that work in parallel. Huge differences between sequential and parallel models lead to appearance of several parallelism stages: algorithmic, hardware and programming stages. It is necessary to reach demanding performance on each stage for development of effective program. Current parallel programming models provide several technologies; within the most popular ones are: OpenMP and MPI. Quite recently parallel technologies were replenished by General-purpose graphics processing units (GPGPU) technique. Each parallel programming model has its own special features. To attain high performance, parallel software should use several levels of parallelism within different models also taking into account its features. But it can be possible to build a programming technique when solving continuum mechanics problems.

2 Two-Level Programming Model for Continuum Mechanics Problems

In spite of all its variety, many computational mathematical models have common details:

– implementation of conservation laws (impulse, mass, energy)
– spatial discretization (construction of structured or unstructured grid)
– governing equations can be written in divergent form:

$$\frac{\partial \bar{Q}}{\partial t} + div\bar{F} = \bar{H}$$

Usually vectors $\bar{Q}$, $\bar{F}$ and $\bar{H}$ are called as vector of the conservative variables, ux vector and the source term, respectively.

– the discrete model is completely defined by a grid function — the state vector $\bar{f}$ function. For example: $\bar{f} = \bar{f}(\bar{x}, t) = (\bar{Q}, \frac{\partial \bar{Q}}{\partial \bar{x}}, \dots)$
– the equations on the state function are commonly local, i.e. link the values at neighboring in space and time grid points.

We can assign a special vector of state to every cell or node that contains solution implicitly or explicitly and other calculated values; calculating algorithm is an iterative modification of state vectors field. Actually, solver based on these principles can be written in program code as a sequence of loops by grid cells with execution of some functions demanding states of current cell and its neighbors. Many actual methods fit into this approach, for example, explicit or implicit finite element method and finite volume methods based on LU-SGS, GMRES, Jacobi methods and many others. This article lies in development of general approach in order to create effective two-level parallel program for solving continuum mechanics problems in above assumptions.

2.1 Levels of Parallelism

On the first level of parallelism our problem is divided into subproblems that are not completely independent and can exchange data between each other. The grid is partitioned by subdomains (Fig. 1). Each subdomain is subtask in the above terms. The graph partitioning problem is very actual NP-hard problem and there are many exact and approximate methods to solve it, for instance, KernighanLin algorithm [1] or balanced graph partitioning algorithm, described in [2].

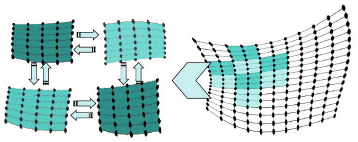


Fig. 1. Area decomposition

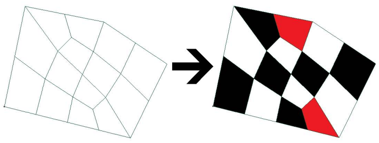


Fig. 2. Example of coloring

The standard well-known trick for computing on several nodes is appending a halo of ghost cells and transfering data from border cells to the ghost ones.

The second level of parallelism is based on multithread technology and some elements are calculated quite simultaneously. In order to avoid collisions we try to separate the cells by several "colors" so that the cells of one color are not adjacent with the cells of another (Fig. 2). It is precisely the graph coloring problem. Then the execution of calculating algorithm sequentially by colors permits us to escape the race condition. The problem to find the chromatic number and associated coloring is NP-hard problem and various exact algorithms have exponential-time rate.

Using the approximate methods, for example greedy coloring algorithm or methods based on backtracking described in [4] and [3], we get the colored grid as a result. To obtain natural black and white coloring as on a chessboard on structured grid we need to define the order of cells at the beginning so that every cell has adjacent predecessor cell except for the first cell chosen randomly.

2.2 Calculating Algorithm

Total calculating algorithm inside each subgrid is divided into iterative execution of the following steps:

1. globally (one for every subgrid) choose "unprocessed" color.
2. process all the border cells of chosen color.
3. transfer processed border cells (updated components of state vector) to adjacent subdomains.
4. process all inner cells of chosen color.
5. synchronize ghost cells state vectors (receive transferred data from adjacent subdomains).
6. end if all colors are processed else return to item 1.

This algorithm uses the advantage of non-blocking internode data transmissions. For instance, asynchronous sending and receiving can be used for MPI. If it is impossible, the items 3 and 5 are combined.

2.3 Memory Distribution

Several specific features of GPGPU have great influence on computing efficiency such as branching, a big number of local variables exceeding the available size of fast registered memory, frequent access to global memory. Overuse of global memory can become a dominating factor for efficiency decline. There are the patterns for access to global memory when several accesses can be coalesced.

We arrange cells state vectors structure as linear array in memory so that memory is sorted by components, each block of components is sorted by cell color, each block of color is sorted by cell type (border, inner, ghost). This trick guarantees that processed data will be placed into memory in immediate proximity, so it leads to better coalescing and more effective cache usage. Also, because of such memory arrangement, we can transfer necessary data during internode synchronization fast. The result memory distribution is shown in Fig. 3.

2.4 Programming Interface

On the one hand, the above approach is independent of the concrete solver and can constitute a common program library. On the other hand, a solver and a problem statement are combined into an interchangeable part of code. Data access and management are controlled by common part through a program interface containing a set of methods to obtain and store a state vector of an arbitrary cell, its' geometry and neighbors' data. The basic solver function can be represented as a function depending on a grid cell iterator.

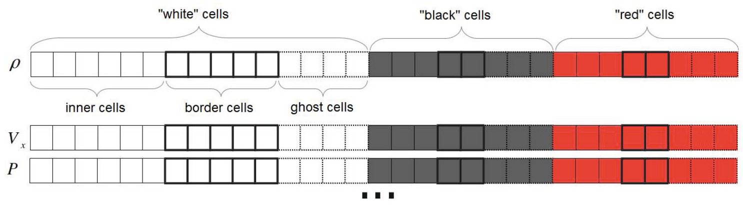


Fig. 3. Example of memory distibution

```
function SomeSolver ( iterator iCell ):
  F := getCellStateVector ( iCell )
  G := getCellGeometry ( iCell )
  ... some operations with F and G ...
  foreach iNeighbor from getCellNeighbors( iCell ):
     Fn := getCellStateVector ( iNeighbor )
     Gn := getCellGeometry ( iNeighbor )
     ... some operation with F, Fn, G, Gn ...
  end foreach
  ... some operations
  setCellStateVector ( iCell, F )
end function
```

The interface solver execution function undertakes parallel sweeps by iterable grid cells with correct data synchronization. Therefore calling the function *RunSolver (SomeSolver)* leads to grid updating to the next iteration.

3 Results

As the result, program package was developed using C++, MPI and CUDA. This package is based on the above propositions for two-dimensional case and has the following distinctive features of implementation:

– independence from concrete mathematical and physical solver (kernel).
– clean and clear programing interface.
– support of calculations both on CPU and GPU, possibility of target architecture fast change.
– support of internode calculations for both architectures using MPI.
– the incapsulation of implementation part.

This package was tested on several different solvers and problems in order to verify the above approach and measure efficiency. LU-SGS solver for hydro and gas dynamics using ideal gas model, LU-SGS solver for multicomponent medium and the simple heat equation solver were developed and integrated to the package.

The performance results for two parallel levels are demonstrated in Fig. 4, 5. The benchmarking tests are executed for parallel levels separately with variation of cell number in mesh.

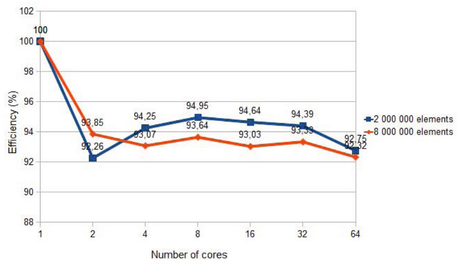


Fig. 4. Efficiency

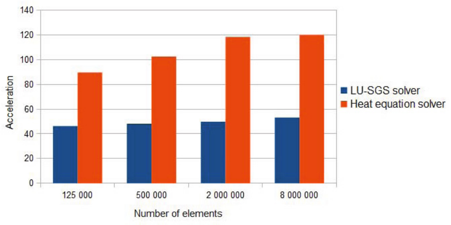


Fig. 5. Acceleration of GPGPU

4 Conclusion

In this paper, the basic approach to construction of effective parallel program for calculation of continuum mechanics problems was developed. The method is based on several fundamental principles and, generally, does not depend on concrete mathematical and physical models. It uses two-level programming model, so some efficiency aspects for parallel programming were described.

Finally, the program package was developed as implementation of the above approach. These package and approach were verified on different physical models and problems and demonstrated good performance results.

References

[1] Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell Systems Technical Journal 49, 291–307 (1970)
[2] Andreev, K., Racke, H.: Balanced graph partitioning, Theory Comput. Systems 39 (2006)
[3] Rodionov, V.: Four-coloring methods for plane graphs (2005) ISBN: 5-484-00127-7
[4] Welsh, D.J.A., Powell, M.B.: An upper bound for the chromatic number of a graph and its application to timetabling problems. The Computer Journal 10(1), 85–86 (1967)

On-Line Parallelizable Task Scheduling on Parallel Processors

Marina Khludova

St-Petersburg State Polytechnic University, Russia
Institute of Applied Mathematics and Mechanics
Department of Telematics
mvkhludova@rambler.ru

Abstract. In a parallelizable task model, a task can be parallelized and the component subtasks can be executed concurrently on multiple processors. The problem of on-line scheduling a stream of independent parallelizable task on a set of parallel identical processors is considered. The processors service the subtasks independently. An analytical model approach based on continuous-time Markov chain is proposed. We study finite capacity queuing model by obtaining numerical expression for the sojourn time. The goal is to choose a technique for evaluation of a resource allocation policy.

Keywords: Task Scheduling, Parallel Identical Processors, Sojourn Time.

1 Introduction

This work was primarily motivated by applications in the area of multiprocessor computer systems running independent parallel programs. The problem of efficient malleable tasks stream scheduling on a set of parallel identical processors (PIP) has received significant attention [1]. In this paper we discuss non-preemptive, non-idling on-line task scheduling and allocation policies, when the actual processing time is usually disclosed upon task arrival. We aim at investigation of allocation policies that can be implemented to actual systems to improve the average system performance.

2 Model Description and Problem Formulation

In this section we present the parallel queuing system used in our study, and provide some technical preliminaries. With the purpose of task scheduling on PIP, we need to take a broader view of the scheduling function as a resource management. This resource management is basically a mechanisms and policies used to efficiently and effectively manage the use of resources by various consumers [2].

2.1 System Model

Consider a parallel processing system comprised of *m* identical processors $P = \{P_1, \ldots P_m\}$ and equipped with an finite capacity queue. The tasks arrive at the system with the Poisson rate λ. The tasks in *T* are independent. Each task processing requires at least one of the processors in *P* and its component subtasks can be processed simultaneously by any number $(\leq m)$ of processors in *P*. If $r \in \{1, \ .. \ m\}$ is the number of processors assigned to task T_j , the processing time of T_j is denoted as $t_j(r)$. We assume that the function $t_j(r)$ is decreasing. The processors independently and concurrently service the subtasks based on a first–come-first-serve (FCFS) discipline. All subtasks can be processed by any processor.

2.2 Investigating Policies

Tasks are treated as statistically identical and scheduled FCFS, since workload characteristics are assumed to be unknown to the scheduler. We assess greedy policies. Initially scheduler assigns the entire set of PIP to the task if it is the only task in queue waiting for service and entire system is free. If two or three tasks are waiting for the service, and the currently finishing task was allocated at the entire set of PIP, the first two tasks in FCFS queue are each allocated at the half of the PIP-set. When a task executed at half of the PIP-set is completed and there are tasks waiting for service, the first task in the queue is scheduled on the released partition of the PIP-set. If four tasks are waiting for the service, and the currently finishing task is allocated at the entire PIP-set, the first four tasks in FCFS queue are individually allocated at the quarter of the PIP-set. When a task that is executed by quarter of the PIP-set completes and there are tasks waiting for service, the first task in the queue is scheduled on the released partition of the set.

3 Markov Analysis

In this section, a technique to construct and compare allocation policies is presented. The policies are modeled using continuous-time Markov chains (CTMC) and performance results are obtained by solving the global balance equations.

3.1 The Workload Characteristics

Differences in parallel applications are taken into consideration by means of workloads with different speedups or, equivalently, processing time function $t_j(r)$. The offered workload is changed (by varying the task arrival rate λ) over the range [0, 1]. It is assumed that inter-arrival and service intervals are exponentially distributed.

3.2 Resource Allocation Decision Process

To calculate and compare the resource allocation process we chose the technique of CTMC. Our primary objective is to calculate the probability distribution of random variable *X(t)* over the space *S*, as the system settles into a regular pattern of behavior (the steady state probability distribution). For small Markov process we use the state transition diagram as the simplest way to represent the process. In this diagram we describe each state of the process as a node in a graph. The arcs in the graph show possible transitions between process states. The arcs are labeled by the transition rates between states. Since every transition is assumed to be governed by an exponential distribution, the rate of the transition is the distribution parameter (Fig.1.).

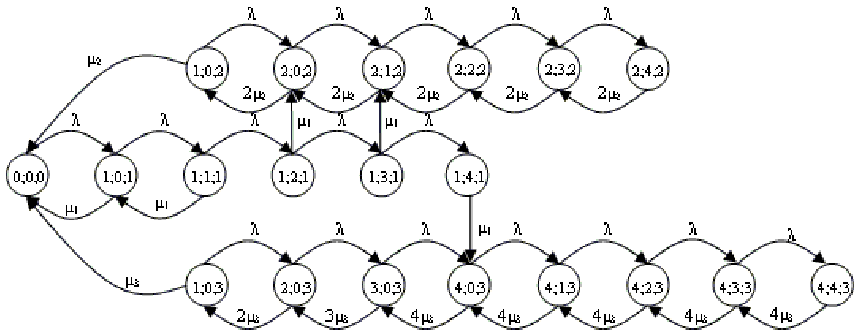


Fig. 1. State transition diagram

The tasks arrive at the system with the Poisson rate λ. States in the model are labeled as *(i,j,k)*, where *i* denotes the number of tasks executed simultaneously on PIP-set, *j* denotes the number of tasks waiting for execution, *k* denotes the type of execution. Type *1* means that only one task is executed on the entire system. Type *2* means two executed tasks on the PIP-set, and type *3* means four executed tasks on the system. As an example, state *(2, 2, 2)* indicates two executed tasks on the PIP-set, and two tasks waiting for service. The state *(0,0,0)* indicates a completely idle system. The number of queued tasks is limited.

The performance results may be derived from solving the global balance equations of Markovian models [3]. When the model is in steady state, we must assume that the total flux-out probability is equal to the total flux-into probability. We denote $\pi(i,j,k)$ as the steady-state probability of the model being in the state $(i,j,k) \in S$. For the state *(0,0,0)* we obtain :

$$\lambda\, \pi(0,0,0) = \mu_1 \pi(1,0,1) + \mu_2 \pi(1,0,2) + \mu_3 \pi(1,0,3),$$

where the input parameters are

λ -the task arrival rate, μ_1 – the average execution rates on entire PIP-set,

μ_2 – the average execution rates on half of PIP-set,
μ_3 – the average execution rates on quarter of PIP-set.

We can involve a weighted sum of steady-state probability to get the average number of tasks in a system and the average sojourn time (applying Little's law):

$$N_s = \Sigma(i+j)\ \pi(i,j,k) \text{ for all states } (i,j,k) \in S.$$

We also can estimate availability of the system by calculating of the probability that incoming task is rejected (Table 1).

4 Performance Evaluation Results

The performance figures obtained from the solution of the global balance equations of Markovian models are reported in this section. The average sojourn time (AST) under a proposed policy is assessed. In this system model we show the effect of

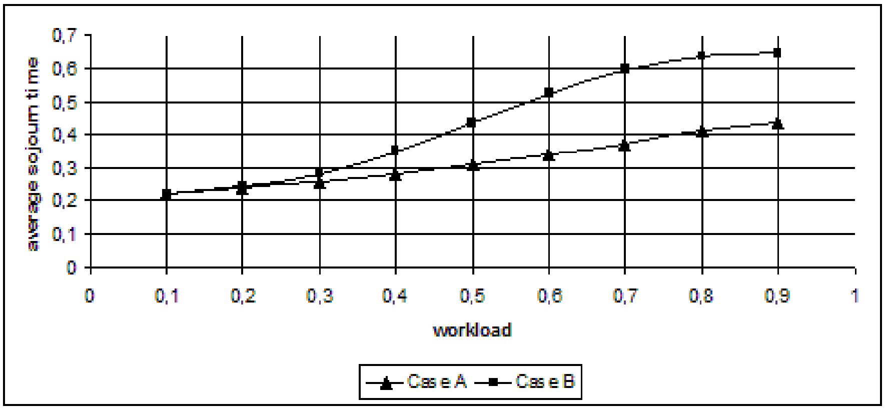


Fig. 2. AST under proposed policy

an offered workload changes (Fig.2) and assume that for case A: $\mu_1 = 16\mu$, $\mu_2 = 8\mu$, $\mu_3 = 4\mu$, case B: $\mu_1 = 16\mu$, $\mu_2 = 16\mu/3$, $\mu_3 = 16\mu/6$, where μ – the average execution rate on a single processor.

Table 1. Task rejection probability

Work load	0.4	0.5	0.6	0.7	0.8	0.9
Case A	0,0033	0,0088	0,0197	0,0379	0,0646	0,0995
Case B	0,0123	0,0390	0,0878	0,1531	0,2236	0,2911

We evaluate the maximal speedup obtained by an ideal PIP-set using examined resource allocation policy. This estimation is based on CTMC and the solution of the global balance equations. Next, we propose some technique to quantify the effect of speedup variation.

5 Conclusion

There are many ways to measure the performance of a parallelizable tasks running on PIP-set. In this paper we have presented CTMC for performance evaluation of resource allocation policy. Currently we have received following results:

(i) a technique to construct CTMC and compare allocation policies has been chosen,
(ii) the effect of variations in offered workload and speedup is quantified.

The main direction for further investigation is to apply the results in continuous-time Markov decision processes [4] to the proposed above technique.

References

1. Monte, J.D., Pattipati, K.R.: Scheduling parallelizable tasks to minimize make-plan and weighted response time. IEEE Transactions on Systems Man and Cybernetics, Part A-Systems and Humans 32, 335–345 (2002)
2. Khludova, M.: Operating systems. Scheduling and Dispatching of Processes. St. Petersburg State Polytechnic University, St. Petersburg (2010) (in Russian)
3. Kleinrock, L.: Queuing Systems, vol.1. Wiley Interscience (1975)
4. Guo, X., Hernandez-Lerma, O.: Continuous-Time Markov Decision Processes, Stochastic Modelling and Applied Probability, vol. 62. Springer, Heidelberg (2009)

Organizing of Parallel Processing User-Friendly Dataflow-Oriented Environment for User Tasks Execution on Cluster

Vladimir Levin, Dmitry Kharitonov, and Daria Odyakova

Institute of Automation and Control Processes, FEBRAS,
Radio 5, 690041 Vladivostok, Russia
`{levin,demiurg,darlene}@dvo.ru`
`http://www.iacp.dvo.ru`

Abstract. In the paper we propose an approach for organizing user tasks on high performance computing systems of cluster type based on dataflow control. This mechanism helps to represent clusters as data file manipulation automata so every user task on remote supercomputers may be described from the point of their input and output files and meta data of these files. Also we present a particular description of a user task file processing mechanism for the system review both from metainformation point of view and dataflow point of view.

Keywords: cloud computing, distributed systems, Petri nets, high-performance systems, dataflow, workflow.

1 Inroduction

During the last two decades supercomputers of cluster type have spreaded all over the world. Users of such systems are mostly scientists and researchers that use clusters for computational experiments modeling physical and chemical processes. Usually each user has a very static set of task kinds, and task execution for each kind differs only by some of parameters. Each user task is a computational process that requires input data, usually as files, and produces output data, usually as files too. A lot of computational software packages use input files, prepared in a special way by additional much less resource consuming tools. On the other hand, output data files obtained in computational experiment also require additional processing to generate diagrams, plots, pictures, movies and other representations suitable for human consumption, for publications and conferences. So computational experiments on clusters could be generally presented with the following three steps. In the first step initial data preprocessing outputs data files. This step is performed as a simple cluster task executed on one node. In the second step these data files are crunched in clusters as a complex user task producing output files or structured groups of files. Finally, in the third step, output files are transformed into a consumable format. From such a schematic view on file processing follows that each file can be used with different kinds of tasks as input to cluster task or as output from cluster task or play both roles.

Traditionally, tasks are chained into complex scientific processes by so-called Scientific Workflow Management Systems (SWFMS) like Taverna, Kepler, Triana, Pegasus, Swift etc. Whereas scientific processes tend to be dataflow oriented[1], SWFMS usually produce workflows to orchestrate cluster activities in terms of Directed Acyclic Graphs (DAG) or script-like languages[2]. From our point of view both approaches have common shortcomings. Workflow is continuous process based on input files provided to the system before execution and without the intrinsic possibility for users to influence on it during execution. Dataflow, however, could be considered as descrete process consisting of steps, each of them is executed by input data availabile with or without user interference depending on the situation at hand. So dataflow process is more flexible with regard to applicability to different patterns of data transformation.

There was quite a lot of successful projects about data flow modelling in terms of Petri nets, and only a few SWFMS based on dataflow without generating workflow[2]. This article presents our vision of a user friendly interface to clusters, where clusters are considered as data file manipulating automata, that produce and execute cluster tasks based on dataflow templates generated using Petri nets and file metadata, both registered in a database. Also two examples with real task file manipulating template descriptions are presented.

2 Overview of the WBS

The WBS (Web Batch System) is designed for the management of user tasks on supercomputers and clusters. The main purpose of the WBS development was providing convenient and friendly environment to create, edit, execute and monitor single-user jobs. Jobs and their files creation is based on templates provided by the WBS [3,4,5]. Templates for jobs and files have access rights based on owner and user-groups. After a job has been created, it starts its life cycle, which consists of such stages as CREATING, EDITING, PENDING (waiting for execution), RUNNING and DONE. The implementation of the WBS has client-server architecture with main components as follows. User access to the WBS is provided by Web-server via internet browser and by special client program for managing job templates and file templates via TCP/IP. All data the WBS deals with is stored on an SQL-server, that also generates a job's options and simple data files. The interface unit between the SQL server and clusters is provided by so-called MODERATOR component, whose main function is to obtain tasks from the SQL server and transfer them as files to appropriate cluster control component named WORKER. The WORKER component reads the task instructions file and prepares directories, copies files, executes programs as it specified in the file. Each WORKER connects to the MODERATOR by TCP/IP and periodically executes a special task to deliver jobs and cluster state to the SQL server.

3 Dataflow Interface from User Point of View

The WBS system is designed for processing user data files. It is assumed that user files are placed in directories inside user-defined locations of cluster-accessable file systems, which the WBS system periodically checks through. All locations to be checked are registered in the database, along with name templates used by the WBS to distinguish directories subjected to examination. The user constructs a new dataflow process by creating a directory in a location registered in the WBS and the next scan of directories will supply the database with information about new directories. A dataflow process type can be sent to the system by placing a *dataflow.wbs* file in the directory, or via WBS interface forms. Dataflow process data is grouped in its directory and directories nested inside it.

Files inside a dataflow process directory should be assigned a type, so that the process will know how to handle this file. Types can be appointed interactively via WEB forms of the WBS system. In the initial stage file types can be specified in the file *dataflow.wbs*, or types can be defined automatically by comparing file names with template file names registered in the database. The next case of file type assignment happens when files are created in the chain of dataflow file transformations, with types defined in the dataflow process task template. The last case of assignment is related to the WBS task execution, when files are created from a database template required for the execution.

Besides the type, files have additional information about the state of their data and metainformation required by the dataflow process. First registration of a file in the WBS system assigns to it the state "initial". Then files are processed by the executing cluster task. After the task completes file state and metainformation can be changed in accordance with the dataflow process template. Task execution can be initiated automatically when a task has all necessary information and files for its execution and its "automated execution" flag defined in the database. Most frequently such situations appear in data post-processing, after the main computational phase of an experiment. When a task has all necessary files for execution but not enough metainformation its execution can be initiated via WBS interactive forms by filling all required metadata fields. This metadata will be attached to the task files and can be used in next steps of the dataflow process. Metainformation can also arise from the results of a task execution by parsing output files with a simple syntax structure.

During the execution or after the execution the data files that are obtained may be transformed to convenient visual representation: images, videos, texts, tables and etc... Those files are uploaded to the database and then available in the graphical user interface of the WBS system. For long-term computation such files are attached to the executing task and can be used by the user to determine whether computation should continue, have its parameters changed, or be terminated immediately. Of course computational tasks should be programmed for periodic checks of refined parameters in advance. Thus the system acquires interactive features: computational tasks provide information for analysis and the user can provide feedback to continuing experiments by changing their parameter files.

4 Interface Implementation

Let us consider the main aspects of the dataflow implementation in the WBS: description and representation of data and metadata in the system and dataflow process in terms of files, parameters and task execution cycle in the system.

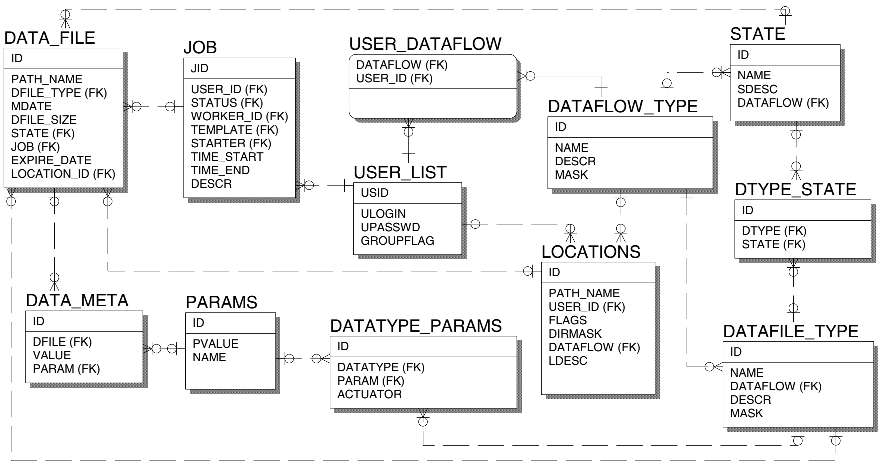


Fig. 1. Metadata in the database ER-diagram

Fig. 1 shows a database entity relation scheme representing the distribution of information about user data and metadata between tables. There is common shared dataflow process types list in the table DATAFLOW_TYPE. Each user from the table USER_LIST has his own sublist of dataflow process types he can run in the table USER_DATAFLOW. In the table LOCATIONS users may register cluster file system directories where they can store further experiments data and directories for each experiments data too. Dataflow process types have a list of data file types in the table DATAFILE_TYPE. User data files in the table DATA_FILE have their types, locations and states from the table STATE. They are associated with completed or still executing cluster task from the JOB table. Besides the data inside the files, there can be also metadata attached to them in the table DATA_META. Metadata is represented as a set of name-value pairs with names from the table PARAMS. A data file can have metadata params, that are registered for their file type in the table DATATYPE_PARAMS.

The database entity relation scheme in Fig. 2 represents interconnections between the tables describing dataflow process templates. Three tables in the scheme connect dataflow definition with the WBS task preparation and execution subsystem: JTEMPLATE, PARAMS and FTEMPLATE. And four tables from Fig. 1: DATAFLOW_TYPE, STATE, DATAFILE_TYPE, DTYPE_STATE link dataflow process templates with data files in hand. Dataflow templates are constructed as coloured Petri nets, where places of each place in Petri net are represented by a tuple of a

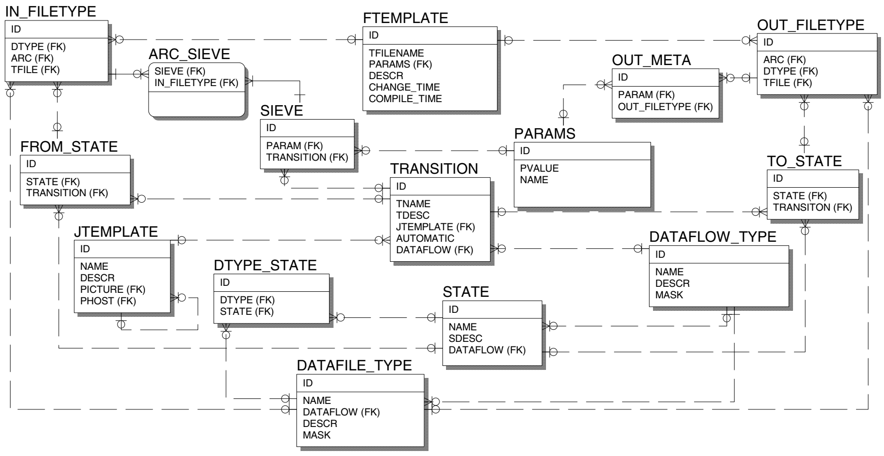


Fig. 2. Dataflow representation in database ER-diagram

data file type and a state. Petri net transitions are stored in the TRANSITION table, which associates with JTEMPLATE. Arcs outgoing from and ingoing to places are represented by records in the tables FROM_STATE and TO_STATE. Tokens of Petri net are represented by data files with their metadata in tables DATA_FILE and DATA_META from Fig. 1. For a transition to be fired all its ingoing arcs from the table FROM_STATE must have a list of data files that matches by type with the types of corresponding list in the table IN_FILETYPE and must have an appropriate state written in the FROM_STATE table. Data files enabling a transition must be of the same dataflow process instance, and parameters indicated in the table SIEVE for the transition must have the same value for all data files in ingoing arcs having file type mentioned in the table ARC_SIEVE. Because all data files for job template are selected by transition, only simple parameters can remain undefined. These parameters are searched for in the file metadata and if all of them are found the cluster task can be executed. After the cluster task is completed, files corresponding with the FTEMPLATE table are updated in database. They obtain the new state from the table TO_STATE and file metadata is updated according to the table OUT_META, assigning values to meta params of files specified in the table OUT_FILETYPE from a completed task.

So each time a transition of Petri net fires it executes a new task on the cluster. This task gets its parameters either from file metadata or directly from the user via interactive forms. A transition results an update of the old file metadata or the registering of new files in the database. This step can be repeated until all files in a dataflow process eventually reach their final state.

5 Examples

Atomic Scale Materials Modelling User Task. The example of user task in this section concerns atomic scale materials modelling. The main computational

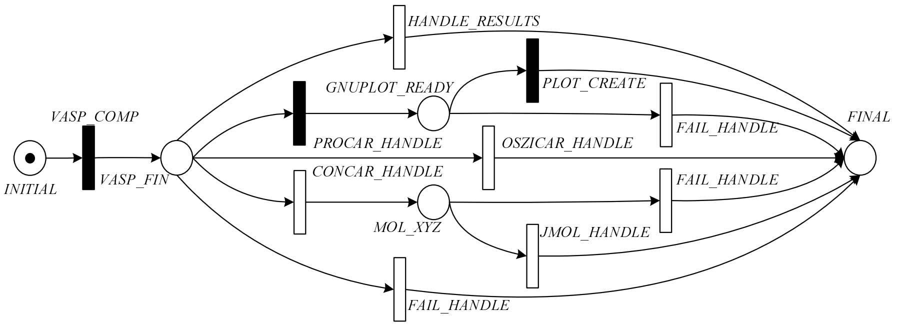


Fig. 3. Schema for VASP concern dataflow

tool for this task is the program package VASP[6]. Users using this tool should, before starting cluster computation, prepare the proper input files and, after computation, need to treat the output files to get images, tables and plots for modelling process.

The main input files are: INCAR, which contains the values of the necessary computation parameters; KPOINTS, which contains the values for the integration scheme description parameters; POTCAR, which contains the pseudopotentials description in a special format; and POSCAR, which contains the description of the computation geometry. The main *VASP* output files are: CONCAR, which contains the molecular dynamics actual coordinates; OSZICAR, which contains a copy of stdout; and PROCAR, with static calculations containing energy concerned values. Every output file should be handled in its own way. For example, *JMOL*[7] is used for CONCAR and *gnuplot*[8] that generates jpeg-files with plots. Within a described task there is a set of states and a set of transitions for dataflow as presented in Fig. 3.

In Fig. 3 circles represent dataflow states and rectangles represent transitions, each of them moving tokens from one state to another one if all criteria for transition are satisfied. Transition satisfaction means that all input files are ready and have appropriate metainformation. For instance, for one of the user DATAFLOW_TYPE that contains a task to get energy plots from *VASP* computation results in jpeg-files. The transitions that correspond to that DATAFLOW_TYPE are solid rectangles shaded in Fig. 3.

Every dataflow process starts in the INITIAL state. To fire the VASP_COMP transition all necessary input files should be ready and have appropriate metainformation. After this transition the dataflow process moves to the VASP_FIN state. Depending on the output files and their metainformation the next proper transition is chosen and fired. For DATAFLOW_TYPE in Fig. 3 if there are no any errors the PROCAR_HANDLE transition is fired. The result for this transition is a file having gnuplot acceptable format. When successful, the dataflow process fires the PLOT_CREATE transition to get output jpeg-files with plots and the dataflow process moves to the FINAL state. This state indicates that the dataflow process is finished.

Gas flow through porous media task. Let us consider the second example for "2D gas flow through porous media" task. The main computation task is a standard grid computation task. The dataflow process for this task is shown in Fig. 4.

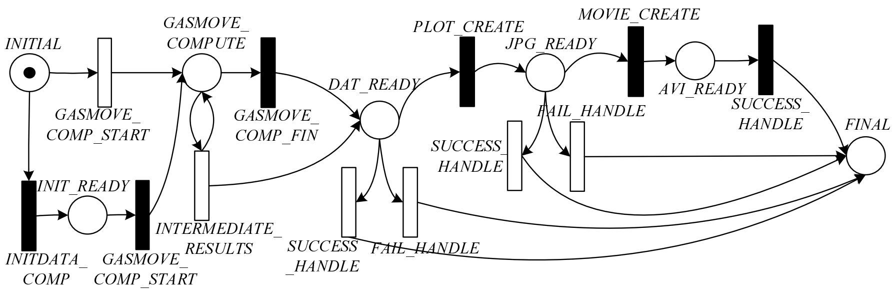


Fig. 4. Schema for gas moving modeling concern dataflow

In this figure circles mean dataflow process states, rectangles are transitions. The input files for the main computation task should consist of six files with initial data for such physical quantities as gas temperature, gas pressure, gas density, temperature of the solid phase, vertical and horizontal filtration velocities. These input files can be generated by special program (corresponding with the INITDATA_COMP transition in Fig. 4) or be gotten from a previous computation. Each file has a metainformation, the most important of which are the following parameters: the computation grid dimension, number of iterations recorded in the file, the name of physical quantity whose values are written to the file (e.g. file dan_p.dat contains values for gas pressure). In Fig. 4 solid transitions represent the user DATAFLOW_TYPE that contains a task to get videos for every physical quantity. Before starting the main computation task (corresponding with the GASMOVE_COMP_START transition in Fig. 4) WBS checks whether all input files are ready and have appropriate metainformation. After the end of main program computation (corresponding with GASMOVE_COMP_FIN) or after writing the intermediate results (corresponding with INTERMEDIATE_RESULTS) the dataflow process moves to the DAT_READY state. Then the PLOT_CREATE transition should be fired. Successful outcomes produce output jpeg-files for every physical quantity. These ones are organized in a structured group of files with the mask *_*.jpg where the first star stands for the number of iterations and the second star stands for the name of corresponding physical quantity. After obtaining a properly ordered group of jpg-files, the MOVIE_CREATE transition can be fired from the JPG_READY state with producing, on success, one output file with an *.avi mask for every physical quantity (e.g. file dan_t.avi contains a video of the gas temperature changing for modeling process). After that the user DATAFLOW_TYPE under review transfers to the FINAL state.

6 Conclusion

This article proposes a new type of interface to multiprocessor computing systems, implemented in user tasks controlling system WBS. By analogy with Cloud Computing concepts this interface can be defined as "Cluster as a Service". In the base of the interface the user work process is represented as a data transformation chain (dataflow) that outputs pictures, diagrams, plots directly suitable for user viewing. The graphical user interface offers to the user an annotated list of files, allowing him to invoke the next link of the transformation chain while the system watches the state of every piece of user data, allowing automatic execution of links with fully prepared input parameters and data. Taking into account the generality of the approach to user dataflow settings, the system can be used in a wide range of cluster tasks minimizing the overall number of files transfered to and from clusters and facilitating routine actions required for users to obtain their final information.

Acknowledgments. The work was funded by Far-Eastern Branch of Russian Academy of Sciences (grant No. 12-I-P14-03).

References

1. Lin, C., Lu, S.,Fei, X.,Chebotko, A., Lai, Z., Pai, D., Fotouhi, F., Hua, J. : A reference architecture for scientific workflow management systems and the VIEW SOA solution, IEEE Transactions on Services Computing, vol. 2, no. 1, pp. 79-92 (2009)
2. Zhao, Z., Belloum, A., Wibisono, A., Terpstra, F.,de Boer, P.T., Sloot, P., Hertzberger, B.: Scientific workflow management: between generality and applicability. In: Proceedings of the International Workshop on Grid and Peer-to-Peer based Workflows in conjunction with the 5th International Conference on Quality Software, vol. 0 pp. 357–364 (2005)
3. Odyakova, D.S., Tarasov, G.V., Kharitonov, D.I.: WBS system as enchanced tool to manage computing environment. In: XII All-Russian Conference on High Perfomance Parallel Computations on Cluster Systems, pp. 300–304. N. I. Lobachevsky State University of Nizhny Novgorod Press, Nizhny Novgorod (2012) (in Russian)
4. Golenkov, E.A., Levin, V.A., Kharitonov, D.I., Shiyan, D.S.: Organization for cyclic computation support in FEBRAS GRID-segment. J. Mining informational and analytical bulletin (scientific and technical journal), 12, vol. 18, pp. 225–229 (2009) (in Russian)
5. Babyak, P.V., Tarasov, G.V.: Organization of process for satellite data flow by means of GRID technology. J. Mining informational and analytical bulletin (scientific and technical journal), 12, vol. 18, pp. 170–175 (2009)
6. The Vienna Ab initio Simulation Package, `https://www.vasp.at/`
7. Jmol, `http://jmol.sourceforge.net/`
8. Gnuplot, `http://www.gnuplot.info/`

SeloGPU: A Selective Off-Loading Framework for High Performance GPGPU Execution

Sejin Park, Jeonghyeon Ma, and Chanik Park

Department of Computer Science and Engineering, POSTECH, Pohang, South Korea
{baksejin,doitnow0415,cipark}@postech.ac.kr

Abstract. In general, GPU accelerated GPGPU application results in much higher performance than CPU application. However, to be accelerated by GPU, users should have GPGPU-enabled computation resources like recent GPU or CPU in their local machine. In this paper, we proposed selective GPGPU off-loading framework named SeloGPU. SeloGPU not only supports remote off-loading for GPGPU application but also supports target node selection among multiple GPGPU-enabled computation resources. We also proposed four optimization techniques to reduce additional overhead owing to remote execution. We implemented SeloGPU using OpenCL which is open standard heterogeneous language. The experimental result shows SeloGPU can choose best target node based on the history of execution information. The four optimization techniques reduce ~87% of network transmission overhead.

Keywords: GPGPU, SeloGPU, offloading, remote execution, OpenCL, CUDA.

1 Introduction

General Purpose GPU (GPGPU) is a technology that enables high performance computing application to use GPU as a computation resource. Traditionally high performance computing is conducted using multiple CPUs. However, this traditional way is changing owing to high performance GPU era. According to various GPU venders, the performance of GPGPU is much higher than CPU for parallelized application owing to GPU's multiple cores. Many previous researches have shown that the performance enhancement is more than 100 times. [1, 2] In order to execute a GPGPU application, we need such high performance GPU hardware in the local machine.

Off-loading is a technique that enables an application or a part of code to run in a remote node. If we apply the off-loading technique to the GPGPU, we can execute high performance GPGPU application using remote side GPU.

However, if there are many nodes that can conduct off-loaded remote processing, which node do we choose to off-load? If the off-loading target node has poor computation resource, then the off-loaded execution may result in slower execution than local execution. In this paper, we propose selective remote OpenCL framework that conducts GPGPU off-loading to achieve high performance execution of GPGPU application. The proposed framework chooses best node to execute GPGPU application based on history information and current network status.

In section 2, background and related work are described. Section 3 describes the architecture of the proposed system. Section 4 explains various experimental results, and finally, section 5 provides some concluding remarks regarding this research.

2 Background and Related Work

2.1 Selective Off-Loading for GPGPU

OpenCL [3] is open standard for parallel programming of heterogeneous systems. OpenCL supports CPU fallback feature so the GPGPU application can execute without GPU. Figure 1 shows the result of matrix multiplication using CPU and GPU.

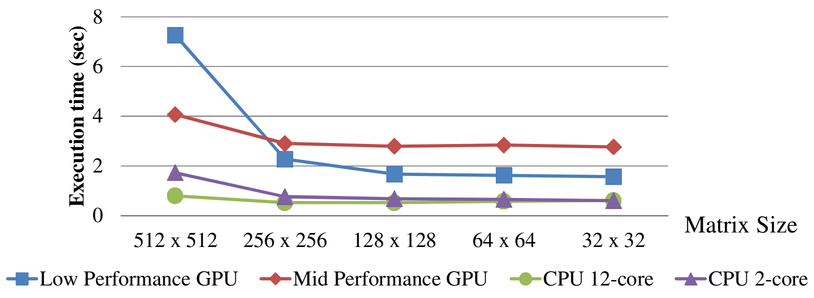


Fig. 1. Matrix multiplication. Intel Xeon E5-2630 is used for CPU experiments. NVidia GeForce 210 and Quadro 2000 are used as low and mid performance GPU, respectively.

As a result, 12-core CPU overwhelms GPGPU for all matrix multiplication sizes and this is the need for selective off-loading for GPGPU application. That is, not all GPU result in faster execution than CPU. We can also think about the case that high network latency with low network bandwidth. In this case, execution in local node would be better even though the remote target has powerful computation resources.

2.2 Related Work

rCUDA [4] supports high speed network transfer using infiniband and their own optimization however details of optimization is not opened since it is not an open source project. It only targets CUDA. vCUDA [5] is remote GPGPU off-loading approach on top of VMM. However, it shows slow results due to the overhead of XML-RPC [6]. dOpenCL [7] proposed uniform programming approach for multi-node-multi-core based OpenCL applications. clOpenCL [8] also proposed distributed execution of OpenCL applications. These two approaches used a wrapper library to interpose the APIs accessing to the OpenCL library. The main differences of these two are the connection method. dOpenCL used TCP based connection and clOpenCL used Open-MX [9] based connection among between OpenCL nodes. These two approaches suggested the method for accessing distributed OpenCL accelerators. However, these two approaches assume that remote accessing will always give better performance than local accessing. Thus, to obtain best performance, users have to select best node and best accelerator for current GPGPU application. GridRPC [10] is a remote procedure call (RPC) mechanism for grid computing. Although it supports grid computing based RPC, this is too general to support GPGPU-oriented optimization. MAUI [11] is an off-loading framework for a network. It off-loads mobile application to the server to overcome battery consumption. Thus their first priority of selective off-loading is energy consumption. It measured energy consumption per API using hardware and it determines whether they will conduct off-load or not. However it requires source code of user application and a special annotation is used to off-load.

3 Architecture

In this section we describe a selective GPGPU off-loading framework named SeloGPU. Figure 2 depicts the architecture of the SeloGPU.

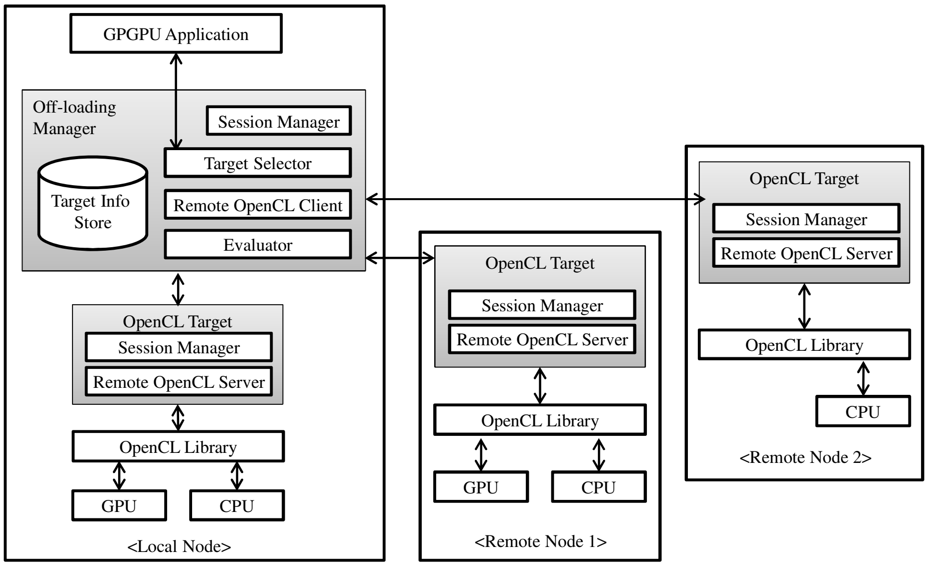


Fig. 2. Architecture of SeloGPU framework

3.1 Off-Loading Manager

A GPGPU application uses OpenCL library to access computation resources. The *Off-loading Manager* interposes between the GPGPU application and the OpenCL

Library. Thus the API access from the GPGPU application is caught to the *Off-loading Manager* without any modification of the GPGPU application.

The *Off-loading manager* consists of five components. The *Session manager* automatically establishes a session with OpenCL Targets since each OpenCL Target broadcasts its advertisement message to the network. When the connection is established, it receives OpenCL Target IDs for computation resources (CPU, GPU) from the Target and calculates current network bandwidth. The *Target Selector* determines a Target that will run OpenCL application with the best performance. The *Remote OpenCL Client* is the actual off-loading engine. It marshals each OpenCL API to the selected *OpenCL Target*'s *Remote OpenCL Server* and receives result of each API. The *Evaluator* measures remote execution time and the output data size of OpenCL application for various *OpenCL Targets* and saves them to the *Target Info Store*. The *Target Info Store* is a simple key-value store. Table 1 depicts this key-value structure.

Table 1. Structure of *Target Info Store*

Key	*Value*
(GPGPU Application ID, OpenCL Target ID)	(Execution time, Output data size)
…	…

GPGPU Application ID is a tuple composed by hash value of kernel source code and the size of input data. The kernel source code and the input data size can be captured by the parameters of OpenCL APIs.

The *Target Selector* chooses best target node for given GPGPU application using stored execution time information in *the Target Info Store*. When a GPGPU application launches, each OpenCL APIs are passed the *Off-loading Manager* and the *Target Selector* chooses best node. Detailed procedure of target selection is as follows.

1. *Target Selector* obtains currently connected OpenCL Target IDs and their network bandwidths from the *Session Manager*. If there is no connection to remote *OpenCL Target*, then it sets local node's *OpenCL Target* as execution target and the *Target Selector* stops. (local execution)
2. If there are one or more connections to the remote *OpenCL Target* then, the *Target Selector* holds OpenCL APIs called from the GPGPU application and just return pre-defined success value until *clCreateProgramWithSource()* calls.
3. When *clCreateProgramWithSource()* calls, it copies the source code to temporary memory and returns pre-defined success value until *clEnqueueWritebuffer()* calls.
4. When *clEnqueueWriteBuffer()* calls, it captures the size of input data.
5. Now we can generate the GPGPU Application ID using hash value of the kernel source code in the temporary memory stored in step 3 and the size of input data.
6. Then it calculates current estimated off-loading time for the GPGPU application with connected *OpenCL Targets* using below equation (1). Note that the output data size and execution time are stored in the *Target Info Store*.

$$\textit{Estimated off-loading time} = \textit{Execution time} + \textit{Network time} \quad (1)$$

$$\textit{Network time} = (\textit{Input data size} + \textit{Output data size}) / \textit{Network bandwidth} \quad (2)$$

7. Finally, the *Target Selector* chooses *OpenCL Target* with the shortest estimated off-loading time.

3.2 OpenCL Target

The *OpenCL Target* runs as a daemon process in the OpenCL target nodes. Both local node and remote node run the *OpenCL Target* because a local node can also be a remote node for another node at the same time. The *OpenCL Target* has two components. The *Session Manager* is responsible for connection to the *Off-loading Manager* in the local side. The *Remote OpenCL Server* unmarshals OpenCL APIs and executes them and sends back the result to the *Remote OpenCL Client* in the local side.

4 Optimizations

4.1 API Pre-execution

Through profiling of the remote execution of the OpenCL APIs, we found that some APIs like *clGetPlatformID()* shows significant higher overhead than the others - about 2.5 seconds. Therefore we can execute them in advance and keep the return value. This optimization results in reducing constant latency which is important for the low-sized computation.

4.2 API Coalescing

The OpenCL APIs called during the initialization stage can be coalesced since the initialization process is always the same. When the bulk data transmission API is called, then the coalesced APIs and the arguments will be sent to the remote node. The bulk data transmission API can be kernel source preparation or input data copy for the kernel. So, OpenCL initialization can be done with a single transmission.

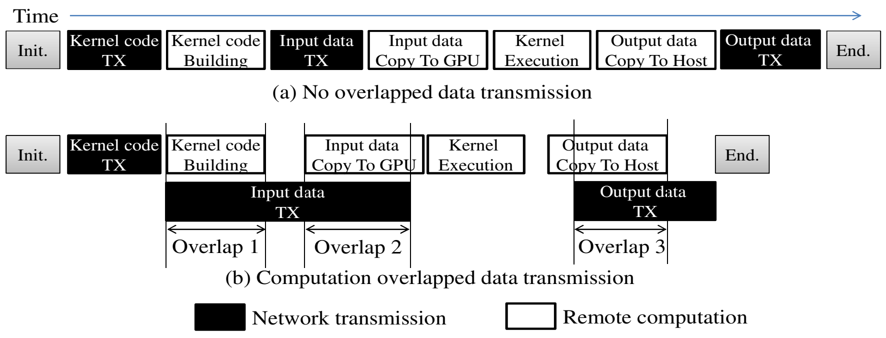


Fig. 3. Overlapped data transmission

4.3 Overlapped Data Transmission

When the OpenCL is initialized, it prepares kernel function and sends input data for the kernel. When the data are completely sent to the remote side, then the kernel function will be executed. In this procedure, we can apply overlapped data transmission since computation and transmission can be existed simultaneously. Figure 3-(a) describes current procedure of data transmission and 3-(b) describes overlapped data transmission. Some loop based GPGPU kernels use their previous output data as current input data. In this case, data transfer from local to host is not needed since the data are already located in the remote side.

4.4 Asynchronous Processing

For APIs that do not need to wait for the return value and if there are no dependency of the API processing, we can convert it to asynchronous calls. APIs in the release stage such as clReleaseKernel(), clReleaseContext() and etc. are those. Therefore, we can process these APIs as asynchronous calls and when these APIs called, then the *Remote OpenCL Client* simply returns to the GPGPU application and sends this call to the *Remote OpenCL Server* at the same time.

5 Evaluation

We used 100Mbps NIC to connect among nodes. The local machine has Intel Atom D510, 1.67GHz CPU with 2GB RAM and NVidia GeForce 210 GPU. The remote machine has Intel Xeon E5-2630 2.3GHz with 48GB RAM and NVidia Quadro 2000 GPU. Two machines run 64bit Ubuntu 12.04 desktop.

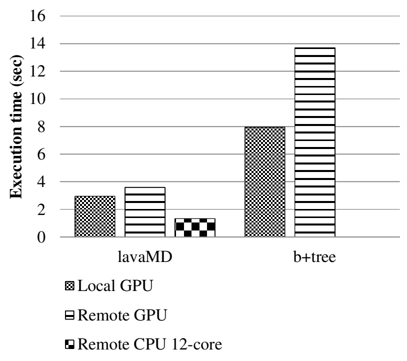


Fig. 4. Off-loading result for Rodinia workload using local GPU and remote GPU and CPU

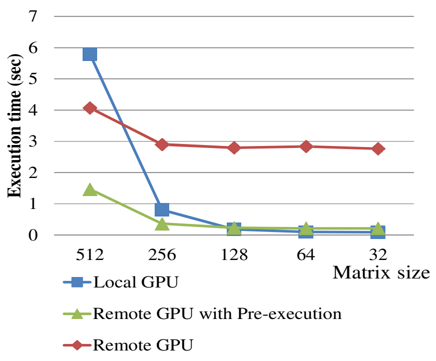


Fig. 5. Result of API Pre-execution

5.1 Selective Off-Loading Result

In order to see the selective off-loading result, we used Rodinia [12] benchmark with one GPU-enabled local machine and two remote machines. (CPU and GPU machines) B+ tree workload in the Rodinia cannot run in CPU owing to a keyword *__synchthread* in kernel function, which is not supported in C99. B+ tree has many internal commands that maintain database and process queries. LavaMD workload in the Rodinia calculates particle position within a large 3D space. Figure 4 shows the result of off-loading result. The *Target Selector* chooses remote CPU 12-core for the lavaMD and local GPU for the B+ tree.

5.2 Optimization Result

Figure 5 shows the result of API pre-execution for matrix multiplication. As depicted in Figure 5, we can save additional about 2.5 seconds by this technique. Especially, this technique fits well with small-sized workload since it reduces constant value. API coalescing, overlapped transmission and asynchronous processing for release API reduce network transmission latency. Table 2 shows the result of each technique.

Table 2. Result of network optimization techniques

	LavaMD	*B+Tree*	*Matrix 256*
API coalescing	0.02%	0.18%	0.19%
Overlapped transmission	1.79%	3.42%	2.04%
Async. processing for release API	0.67%	0.3%	0.65%

Figure 6 shows the result with all optimizations. The lavaMD shows about 80% reduced overhead than the original remote execution, and the b+tree shows about 16% reduced overhead than the original remote execution. Matrix multiplication result shows about 87% reduced overhead. These various results are originated from the API pre-execution. The API pre-execution reduces constant time (about 2.5 sec.) so it depends on the size of the workload.

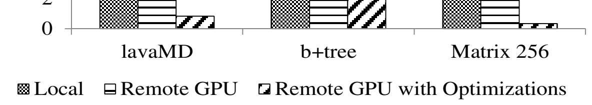


Fig. 6. Result of overall optimization. Matrix 256 means matrix multiplication of 256 x 256.

6 Conclusion

In this paper, we proposed selective GPU off-loading architecture named SeloGPU. We described detailed architecture of SeloGPU. It consists of two parts: the *Off-loading Manager* in local node and the *OpenCL Target* in remote node. Since the off-loading manager interposes between GPGPU application and the OpenCL library, we can achieve off-loading without any modification of the application. Based on the stored execution information in the *Target Info Store*, SeloGPU can choose best node to execute current application. We also proposed four optimization techniques which achieve 16-87% reduced network transmission overhead.

Acknowledgment. This work was supported by the IT R&D program of MKE/KEIT, [K10041801, Zero Configuration Type Device Interaction Technology using Device Sociality between Heterogeneous Devices].

References

1. Gao, W., Huyen, N.T.T., Loi, H.S., Kemao, Q.: Real-time 2D parallel windowed Fourier transform for fringe pattern analysis using Graphics Processing Unit. Opt. Express 17(25), 23147–23152 (2009)
2. Lu, P.J., Oki, H., Frey, C.A., Chamitoff, G.E., et al.: Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station. Journal of Real-Time Image Processing 5(3), 179–193 (2010)
3. OpenCL, `http://www.khronos.org/opencl/`
4. Duato, J., Igual, F.D., Mayo, R., Peña, A.J., Quintana-Ortí, E.S., Silla, F.: An efficient implementation of GPU virtualization in high performance clusters. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 385–394. Springer, Heidelberg (2010)
5. Shi, L., Chen, H., Sun, J., Li, K.: vCUDA: GPU-accelerated high-performance computing in virtual machines. In: IEEE International Symposium on Parallel & Distributed Processing (2009)
6. Simon, L., Joe, J., Edd, D.: Programming Web Services with XML-RPC, 1st edn. O'Reilly (2001)
7. Kegel, P., Michel, S., Sergei, G.: dOpenCL: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In: 21th International Heterogeneity in Computing Workshop (HCW 2012) (2012)
8. Alves, A., Rufino, J., Pina, A., Santos, L.P.: *cl*OpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters. In: Caragiannis, I., et al. (eds.) Euro-Par Workshops 2012. LNCS, vol. 7640, pp. 112–122. Springer, Heidelberg (2013)
9. Goglin, B.: High-Performance Message Passing over generic Ethernet Hardware with Open-MX. Elsevier Journal of Parallel Comp 37(2), 85–100 (2011)
10. Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C., Casanova, H.: Overview of GridRPC: A remote procedure call API for grid computing. In: Parashar, M. (ed.) GRID 2002. LNCS, vol. 2536, pp. 274–278. Springer, Heidelberg (2002)
11. Cuervo, E., Balasubramanian, A., Cho, D.K., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: MAUI: making smartphones last longer with code offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, pp. 49–62. ACM (2010)
12. Rodinia 2.2, `http://www.cs.virginia.edu/~skadron/wiki/rodinia/`

Efficient Domain Decomposition of Dissipative Particle Dynamics via Choice of Pseudorandom Number Generator

Michael Seaton[1], Ilian Todorov[1], and Yaser Afshar[2]

[1] Scientific Computing Department, STFC Daresbury Laboratory, Warrington, Cheshire, WA4 4AD, United Kingdom
[2] Max Planck Institute of Molecular Cell Biology and Genetics, Pfotenhauerstr. 108, 01307 Dresden, Germany
`michael.seaton@stfc.ac.uk`

Abstract. Domain decomposition of dissipative particle dynamics is complicated by the use of random pairwise forces as a component of a momentum-conserving thermostat. The conventional use of a pseudorandom number generator for each processor core leads to the need for an additional communication step to correctly assign random forces to particles in boundary halos. To circumvent this communication, the use of a three-seed pseudorandom number generator is proposed to allow multiple processor cores to evaluate the same forces. This kind of pseudorandom number generator will be applied to the general-purpose mesoscale modelling package DL_MESO to improve its parallel scalability for large processor core counts.

Keywords: Domain decomposition, dissipative particle dynamics, pseudorandom number generator, DL_MESO.

1 Introduction

Dissipative particle dynamics (DPD) is a particle-based technique for modelling systems at the mesoscale[1], often used to study soft matter, complex fluids and biomolecular systems. It is based upon a Galilean invariant thermostat provided by pairwise dissipative and random forces, which correctly reproduces hydrodynamics[2] while allowing the use of larger time-steps than classical molecular dynamics (MD). This allows the use of a smaller number of larger, softer particles to represent a system dominated by fluid dynamics, making the technique capable of modelling systems approaching continuum length and time scales.

Like classical MD, DPD calculations can be parallelized by domain decomposition, which divides the particles among processor cores according to their locations in the volume. This parallelization technique relies on autonomous calculations of particle forces with some communication to deport particles leaving the volume for each core and to create boundary halos for force calculations. An additional complication in DPD is the use of pairwise random forces, which are frequently reliant on core-dependent pseudorandom number generators (PRNG) and thus require additional communication of calculated forces to assign them correctly to particles in boundary halos. This method is implemented in the DPD code of DL_MESO[3,4]; while the parallel scalability of this code is generally very good, it suffers at higher core counts due to increased communication.

To eliminate communications for force assignments, an alternative PRNG can be used that can easily and rapidly produce a random number using the numbers for any given pair of particles and the time step as seeds[5]. Since this PRNG can reproduce the same random number for the same seeds on any processor core, this removes the need to communicate random forces between cores and allows duplicate pairwise force calculations to be carried out on all cores. The objective of this work is to improve the parallel scalability of the DPD code in DL_MESO by incorporating a PRNG with these properties and modifying pairwise force calculations, with the intention of including this feature in future code releases.

2 Theoretical and Computational Background

2.1 Dissipative Particle Dynamics

A DPD system consists of a number of particles whose motion is determined by integrating the forces acting on them due to interactions with each other. The force acting on particle i is given by

$$\mathbf{F}_i = \sum_{j \neq i} \left(\mathbf{F}_{ij}^C + \mathbf{F}_{ij}^D + \mathbf{F}_{ij}^R \right) \tag{1}$$

with $\mathbf{F}_{ij}^C$, $\mathbf{F}_{ij}^D$ and $\mathbf{F}_{ij}^R$ as the conservative, dissipative and random forces between particles i and j. Additional forces can be included for interactions due to bonds, electrostatics, surfaces etc. The conservative force is normally selected to give a soft-core quadratic potential[6] but can take on alternative forms such as hard-core potentials typically used for classical MD[7] and density-dependent potentials[8].

The dissipative and random forces act as the system thermostat. The dissipative force is expressed by

$$\mathbf{F}_{ij}^D = -\gamma w^D(r_{ij})(\hat{\mathbf{r}}_{ij} \cdot \mathbf{v}_{ij})\hat{\mathbf{r}}_{ij} \tag{2}$$

and the random force by

$$\mathbf{F}_{ij}^R = \sigma w^R(r_{ij})\theta_{ij}\hat{\mathbf{r}}_{ij} \tag{3}$$

where γ and σ are the amplitudes of the dissipative and random forces respectively, $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ is the vector between particles i and j, $\mathbf{v}_{ij} = \mathbf{v}_j - \mathbf{v}_i$ the relative velocity between particles i and j, $\hat{\mathbf{r}}_{ij} = \frac{\mathbf{r}_{ij}}{r_{ij}}$, $w^D(r_{ij})$ and $w^R(r_{ij})$ are weighting functions dependent on particle separation and $\theta_{ij} = \theta_{ji}$ is a randomly fluctuating variable with Gaussian statistics, i.e.

$$\begin{aligned} \langle \theta_{ij}(t) \rangle &= 0 \\ \langle \theta_{ij}(t) \cdot \theta_{kl}(t') \rangle &= (\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})\delta(t - t'). \end{aligned} \tag{4}$$

The thermostat is achieved by fixing relationships between the dissipative and random weighting functions and force amplitudes to satisfy the Fokker-Planck fluctuation-dissipation theorem[2]:

$$w^D(r_{ij}) = \left[w^R(r_{ij})\right]^2 \tag{5}$$

$$\sigma^2 = 2k_BT\gamma \tag{6}$$

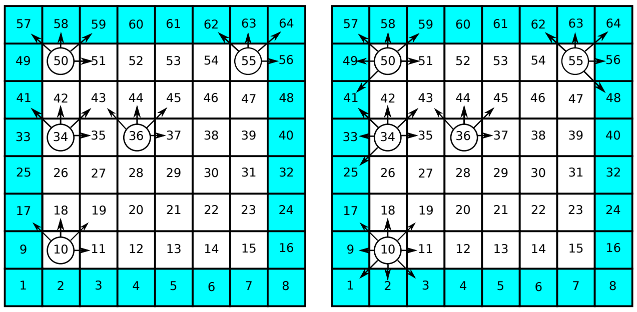
where k_B is the Boltzmann constant and T is the desired system temperature. The weighting function for random forces is normally selected to be $w^R(r_{ij}) = 1 - \frac{r_{ij}}{r_c}$ for $r_{ij} < r_c$, the same functional form as normally selected for the conservative force, while the dissipative force amplitude is related to the viscosity of the fluid.

2.2 Domain Decomposition, Link-Cell Lists and Communication

The DPD code of DL_MESO is parallelized using domain decomposition: the volume of the simulation cell is divided equally among the number of processor cores available, with each sub-volume holding a number of DPD particles. To calculate forces correctly, the construction of a boundary halo with a size equal to the maximum interaction cutoff (r_c) is required to allow for calculations of pairwise forces between particles in neighbouring process cores.

The search for pairs of particles within a cutoff distance can be carried out efficiently by means of link-cell lists[9]: each sub-volume (including its boundary halo) is split into an integer number of cuboidal cells with sides equal to or greater than r_c, the particles can be assigned to those cells and lists of particles in each cell can be constructed. By working through the list of particles for a particular cell and the lists of nearest neighbour cells, pairs of particles likely to be within the cutoff can be found. Re-calculation of particle pairs is avoided by only directly searching cells belonging to the sub-volume and restricting the search of neighbouring cells to half of all possible directions: this is illustrated by Figure 1(a). The search for nearest neighbours can be extended to include all neighbouring boundary cells for cells at the edge of the sub-volume, as illustrated in Figure 1(b), provided the forces between pairs in the additional cells can be reproduced in neighbouring sub-volumes. Highly-scalable classical molecular dynamics codes such as DL_POLY_4[10,11] use this approach.

Domain decomposition is dependent on communication between neighbouring processor cores. A *deport* step is required to move any particles leaving a sub-volume into the appropriate processor core, while an *export* step is needed to construct a boundary halo from particles within r_c of the boundary between sub-volumes. If the half-way search of link-cells is applied, an additional *import* step is required to send forces for particles in the boundary halo back to the processor cores they belong to for summation. No import step is required if a full search of boundary cells is carried out, as all contributions to forces for all particles in the sub-volume will have been determined.



(a) Half-way search pattern (b) Full search pattern

Fig. 1. Illustration of searches for particle pairs in a link-cell algorithm for 2-D simulation. The half-way search (a) uses the same pattern for each of the (white) cells within the volume, while the full search (b) includes all possible boundary (cyan) cells.

2.3 Random Forces

The Gaussian random fluctuations θ_{ij} can be calculated from Gaussian random numbers with zero mean and unit variance, ξ_{ij}, and the time-step for simulations, Δt:

$$\theta_{ij} = \frac{\xi_{ij}}{\sqrt{\Delta t}}. \tag{7}$$

Many pseudorandom number generators produce uniformly distributed random numbers u_{ij} (often between 0 and 1), so some form of transformation is often required. For DPD simulations, it is possible to exploit the central limit theorem and use uniform random numbers directly as an approximation for Gaussian random numbers[6], i.e.

$$\xi_{ij} \approx \sqrt{12}\left(u_{ij} - \tfrac{1}{2}\right), \tag{8}$$

which gives statistically indistinguishable results to true Gaussian random numbers.

For calculations of pseudorandom numbers in parallel computing environments, a commonly-used scheme is to have a PRNG for each processor core, i.e. each core has a uniquely seeded, independent random number stream. This is currently used in the DPD code of DL_MESO, which uses the Mersenne Twister (MT) PRNG[12] with seeds based upon the processor core number. This approach makes it virtually impossible to generate the same random number for each particle pair at a given time-step on multiple processor cores, which thus requires the half-way search of link-cells and the additional import communication step to send forces back to neighbouring cores. The effect of this additional communication becomes more pronounced as the number of processor cores increases, which has been observed to cause a plateauing and decline in parallel speed-up for a system of fixed size[3].

3 Main Ideas

To improve the parallel scalability of DL_MESO's DPD code, a reduction in communication is proposed by eliminating the need for a force importing step, which requires an alternative method of easily and quickly generating the same random numbers independently on separate processor cores for each particle pair at a given time-step. The PRNG *Saru* was selected for this work due to its portability, ease of implementation, rapid seeding and availability of a three-variable seed premixing algorithm.

3.1 Fast Seeding PRNG

Saru, a PRNG written by Steve Worley of Worley Laboratories, is designed to be robust, fast in evaluating pseudorandom numbers and advancing its state, and have a small state size[5]. It has been observed to be faster than the MT PRNG in terms of both seeding and evaluation, and has passed all randomness tests.

The state size of *Saru* is two 32-bit 'words' — `state` and `wstate` — which are advanced by means of a linear congruential generator (LCG) and an Offset Weyl Sequence (OWS):

```
state  = 0x4beb5d59*state + 0x2600e1f7;
wstate = wstate + oWeylOffset + ((((signed int)wstate)>>31)&oWeylPeriod);
```

where `oWeylOffset=0x8009d14b` and `WeylPeriod=0xda879add`. These two components are combined in such a way as to obscure any regular patterns, using simple XORs, bit-shifting and a multiply:

```
unsigned int v = (state ^ (state>>26))+wstate;
return (v^(v>>20))*0x6957f5a7;
```

which produces values between 0 and $2^{32}-1$: this can subsequently be converted into a uniformly distributed random number between 0 and 1. The period of the *Saru* PRNG is $\sim 2^{63.77}$, which can be limiting for systems involving more than a few billion samples but can be extended at the cost of additional storage space and update speed. *Saru* is seeded in every single call and its resulting state is not used in subsequent calls; in this sense, *Saru* can thus be described as state-free and, unlike many PRNGs, no storage of its state is required to resume calculations.

3.2 Three-Variable Seeding

The two 32-bit words used for *Saru* can be generated using one or more seeding values. In the context of DPD simulations, a three-variable seed is of particular interest since the particle indices and time-step can be used to generate the 32-bit words and subsequently produce a unique pseudorandom number. Extensive testing of hundreds of mixing functions with millions of constants produced the following premixing algorithm[5]:

```
seed3 ^= (seed1<<7)^(seed2>>6);
seed2 += (seed1>>4)^(seed3>>15);
seed1 ^= (seed2<<9)+(seed3<<8);
seed3 ^= 0xa5366b4d*((seed2>>11) ^ (seed1<<1));
seed2 += 0x72be1579*((seed1<<4)  ^ (seed3>>16));
seed1 ^= 0x3f38a6ed*((seed3>>5)  ^ (((signed int)seed2)>>22));
seed2 += seed1*seed3;
seed1 += seed3 ^ (seed2>>2);
seed2 ^= ((signed int)seed2)>>17;
```

with `seed1` and `seed2` as the smaller and larger particle global indices respectively and `seed3` as the time-step. The three seeds are then transformed into a two integer state using bitwise XOR, multiplication, addition, bit-shifting and type conversion:

```
state  = 0x79dedea3*(seed1^(((signed int)seed1)>>14));
wstate = (state + seed2) ^ (((signed int)state)>>8);
state  = state + (wstate*(wstate^0xdddf97f5));
wstate = 0xabcb96f7 + (wstate>>1);
```

The resulting values of `state` and `wstate` are then advanced and combined as described above.

4 Results and Conclusions

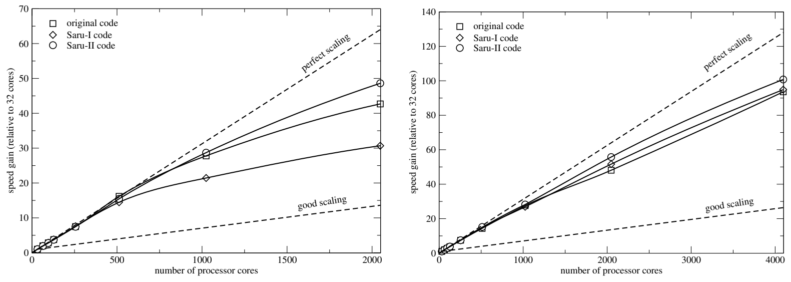
Starting with the publicly released version of DL_MESO version 2.5, the DPD code has been modified twice. One version (Saru-I) implements the *Saru* PRNG with three-variable seeding in place of the MT PRNG but retains the half-way search pattern for pairwise force calculations and the force import communication step, while the other (Saru-II) implements *Saru* and the full search pattern, thus allowing the force import step to be omitted.

The book-keeping required for bonded interactions in Saru-I is left unchanged from the release version of DL_MESO, which relies on calculating each bond once and using the force import step for particles in boundary halos. Elimination of the force import step meant it could not be used unmodified for Saru-II, so a more sophisticated book-keeping scheme – the form used in DL_POLY 4[10] – was included to allow autonomous calculation of bond potentials and forces in multiple processor cores for bonds that span multiple sub-volumes.

Simulations of a simple 3000-particle system with two separating species were carried out in serial for all three codes to verify the statistical consistency of the two PRNGs: respectively for the original, Saru-I and Saru-II versions of DL_MESO, average system temperatures of $k_BT = 1.0194 \pm 0.1109$, 1.0184 ± 0.1111, 1.0176 ± 0.1111 and pressures (in DPD units) of $\frac{pr_c^3}{k_BT} = 25.502 \pm 1.417$, 25.609 ± 1.401, 25.586 ± 1.403 were achieved. Direct measurements of random force calculations for the original and Saru-I codes suggest that *Saru* with the three-seed premixing algorithm takes approximately 6% more time than MT to produce a pseudorandom number, but the use of the full particle pair search pattern in Saru-II to eliminate a serial form of the force import step gives a significant time-saving, reducing the runtime for this system by about 25%.

The parallel scalabilities of the versions of DL_MESO with the *Saru* PRNG were tested using simulations of vesicle formation from solutions of amphiphilic molecules[3,13] on the UK's national high performance computing service, HECToR[14], which consists of 32-core Cray XE6 processors.

The largest strong and weak scaling tests from [3] (therein denoted as S-DPD-III and W-DPD-III respectively) were initially used without bonded interactions to determine the effect of changing PRNGs and force calculations/communication: these are illustrated in Figure 2. Compared with the original version of the code, Saru-I suffers from a drop in strong scalability but its weak scalability is almost unchanged; the most likely explanation is the additional memory access required to determine the global particle indices as seeds for *Saru*. Changing pair searching and omitting the force import step more than compensates for this, however, as Saru-II outperforms the original version of the code in both tests.



(a) Strong scalability (b) Weak scalability

Fig. 2. Scalability of original, Saru-I and Saru-II DL_MESO codes on HECToR for systems without bonded interactions. Strong scalability tested using a 11.4 million particle system and weak scalability using 44686.5 particles per core.

The smallest and largest strong scaling tests from [3], now including bond interactions in equal proportions of particle numbers, were carried out using Saru-II for comparison with pre-existing results for the original code: these are shown in Figure 3. While there is a modest improvement in scalability for the small system using Saru-II, this version of the code does not scale as well as the original version beyond 256 cores for the large system in spite of the previously observed improvements due to the changes in PRNG, particle pair search and communication. It is clear that the form of book-keeping for bond interactions has a significant effect on scalability at higher core and bond counts, particularly in terms of contiguity of this data for memory access.

The use of *Saru* with the three-seed premixing algorithm does significantly improve the parallel scalability of DL_MESO and its inclusion in future releases is strongly justified. Further optimization of DL_MESO would be required to fully exploit the full linked-cell search scheme, particularly improvements to memory contiguity of particle data and book-keeping of bonded interactions.

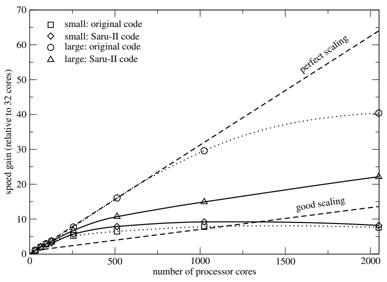


Fig. 3. Strong scalability of original (dotted lines) and Saru-II (solid lines) versions of DL_MESO on HECToR for 715 thousand and 11.4 million particle systems (S-DPD-I and S-DPD-III in [3]) including bonds between particles

Acknowledgments. Thanks are due to Steve Worley (Worley Laboratories, USA) for providing the source code for *Saru*.

References

1. Hoogerbrugge, P.J., Koelman, J.M.V.A.: Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics. Europhys. Lett. 19, 155–160 (1992)
2. Español, P., Warren, P.: Statistical mechanics of dissipative particle dynamics. Europhys. Lett. 30, 191–196 (1995)
3. Seaton, M.A., Anderson, R.L., Metz, S., Smith, W.: DL_MESO: highly scalable mesoscale simulations. Mol. Sim. (in press, published online, 2013)
4. The DL_MESO Mesoscale Simulation Package, `http://www.ccp5.ac.uk/DL_MESO/`
5. Afshar, Y., Schmid, F., Pishevar, A., Worley, S.: Exploiting seeding of random number generators for efficient domain decomposition parallelization of dissipative particle dynamics. Comp. Phys. Comms. 184, 1119–1128 (2013)
6. Groot, R.D., Warren, P.B.: Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation, J. Chem. Phys. 107, 4423–4435 (1997)
7. Vattulainen, I., Karttunen, M., Besold, G., Polson, J.M.: Integration schemes for dissipative particle dynamics simulations: from softly interacting systems towards hybrid models. J. Chem. Phys. 116, 3967–3979 (2002)
8. Trofimov, S.Y., Nies, E.L.F., Michels, M.A.J.: Thermodynamic consistency in dissipative particle dynamics simulations of strongly nonideal liquids and liquid mixtures. J. Chem. Phys. 117, 9383–9394 (2002)
9. Hockney, R.W., Eastwood, J.W.: Computer Simulation Using Particles. McGraw-Hill International, New York (1981)
10. The DL_POLY Molecular Simulation Package, `http://www.ccp5.ac.uk/DL_POLY/`
11. Todorov, I.T., Smith, W.: DL_POLY_3: the CCP5 national UK code for molecular-dynamics simulations. Phil. Trans. Roy. Soc. Lond. A 362, 1835–1852 (2004)
12. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. on Modeling and Computer Simulation 8, 3–30 (1998)
13. Yamamoto, S., Maruyama, Y., Hyodo, S.: Dissipative particle dynamics study of spontaneous vesicle formation of amphiphilic molecules. J. Chem. Phys. 116, 5842–5849 (2002)
14. HECToR: UK National Supercomputing Service, `http://www.hector.ac.uk`

SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems

Michel Steuwer and Sergei Gorlatch

University of Muenster, Germany
{michel.steuwer,gorlatch}@wwu.de

Abstract. Application development for modern high-performance systems with Graphics Processing Units (GPUs) currently relies on low-level programming approaches like CUDA and OpenCL, which leads to complex, lengthy and error-prone programs.

In this paper, we present SkelCL – a high-level programming approach for systems with multiple GPUs and its implementation as a library on top of OpenCL. SkelCL provides three main enhancements to the OpenCL standard: 1) computations are conveniently expressed using parallel algorithmic patterns (*skeletons*); 2) memory management is simplified using parallel *container data types* (vectors and matrices); 3) an automatic *data (re)distribution mechanism* allows for implicit data movements between GPUs and ensures scalability when using multiple GPUs. We demonstrate how SkelCL is used to implement parallel applications on one- and two-dimensional data. We report experimental results to evaluate our approach in terms of programming effort and performance.

1 Introduction

Modern high-performance computer systems become increasingly heterogeneous as they comprise in addition to multi-core processors, also *Graphics Processing Units* (GPUs), Cell processors, FPGA, and other accelerating devices, usually called *accelerators*. The state-of-the-art application programming for systems with GPUs is cumbersome and error-prone, because GPUs are programmed using explicit, low-level programming approaches like CUDA [11] or OpenCL [13]. These approaches require the programmer to explicitly manage GPU's memory (including memory (de)allocations, and data transfers to/from the system's main memory), and explicitly specify parallelism in the computation. This leads to lengthy, low-level, complicated and, thus, error-prone code. For multi-GPU systems, programming with CUDA and OpenCL is even more complex, as both approaches require an explicit implementation of data exchange between the GPUs, as well as disjoint management of each GPU, including low-level pointer arithmetics and offset calculations.

In this paper, we describe the *SkelCL* (Skeleton Computing Language) – our high-level programming approach for parallel systems with multiple GPUs. The SkelCL programming model is based on the OpenCL standard and enhances it with three high-level mechanisms:

1) *parallel container data types*: collections of data (in particular, vectors and matrices) that are managed automatically on all GPUs in the system;
2) *data (re)distributions*: an automatic mechanism for specifying in the application program suitable data distributions and re-distributions among the GPUs of the target system:
3) *parallel skeletons*: pre-implemented high-level patterns of parallel computation and communication which can be customized to express application-specific parallelism, and combined to a large high-level code.

The structure of the paper is as follows. In Section 2 we formulate the requirements to a high-level programming approach for GPU systems, following from the analysis of compute-intensive applications. Section 3 describes in detail our SkelCL approach. In Section 4 we report experimental evaluation of our approach regarding both programming effort and performance. We compare to related work and conclude in Section in Section 5.

2 Requirements to a High-Level Programming Model

To simplify programming for a system with multiple GPUs, the following high-level abstraction are desirable:

Parallel container data types. Compute-intensive applications typically operate on a (possibly big) set of data items. Managing memory hierarchy of multi-GPU systems explicitly is complex and error-prone because low-level details, like offset calculations, have to be programmed manually. A high-level programming model should be able to make collections of data automatically accessible to all GPUs in the target system and it should provide an easy-to-use interface for the application developer.

Distribution and redistribution mechanisms. To achieve scalability of applications on systems comprising multiple GPUs, it is crucial to decide how the application's data are distributed across all available GPUs. Applications often require different distributions for their computational steps. Distributing and re-distributing data between GPUs in OpenCL is cumbersome because data transfers have to be managed manually and performed via the CPU. Therefore, it is important for a high-level programming model to allow both for describing the data distribution and for changing the distribution at runtime, such that the system takes care of the necessary data movements.

Recurring patterns of parallelism. While the concrete operations performed in an application are (of course) application-specific, the general structure of parallelization often follows some common parallel patterns that are reused in different applications. For example, operations can be performed for every entry of an input vector, which is a well-known pattern of data-parallel programming, or two vectors are combined element-wise into an output vector, which is again a common pattern of parallelism. It would be, therefore, desirable to express the high-level structure of an application using pre-defined common patterns, rather than describing the parallelism explicitly in much detail.

3 SkelCL: Programming Model and Library

We develop our SkelCL [14] programming model as an extension of the standard OpenCL programming model [13], which is an emerging de-facto standard for programming heterogenous systems with various accelerators. SkelCL adds to OpenCL three features that we identified as desirable in Section 2. SkelCL inherits all properties of OpenCL, including its portability across different heterogeneous parallel systems. SkelCL is designed to be fully compatible with OpenCL: arbitrary parts of a SkelCL code can be written or rewritten in OpenCL, without influencing program's correctness. While the main OpenCL program is executed sequentially on the CPU – called the *host* – computations are offloaded to parallel processors – called *devices*. In this paper, we focus on systems comprising multiple GPUs, therefore, we use the terms CPU and GPU, rather than more general OpenCL terms host and device.

3.1 Parallel Container Data Types

SkelCL offers the application developer two container classes – vector and matrix – which are transparently accessible by both, host and devices, i. e. the CPU and the GPUs. The *vector* abstracts a one-dimensional contiguous memory area while the *matrix* provides an interface to a two-dimensional memory area. When a container is created on the CPU, memory is allocated on the GPUs automatically; when a container on the CPU is deleted, the memory allocated on the GPUs is freed automatically. In a SkelCL program, a vector object can be created and filled with data as in the following example:

```
Vector<int> vec(size);
for (int i = 0; i < vec.size(); ++i){ vec[i] = i; }
```

The main advantage of the container data types in SkelCL as compared with OpenCL is that the necessary data transfers between the CPU and GPUs are performed implicitly. Before performing a computation on container types, the SkelCL system ensures that all input containers' data is available on all participating GPUs. This may result in implicit (automatic) data transfers from the CPU to GPU memory, which in OpenCL would require explicit programming. Similarly, before any data is accessed on the CPU, the implementation of SkelCL ensures that this data on the CPU is up-to-date by performing necessary data transfers implicitly and automatically. Thus, the container classes shield the programmer from low-level operations like memory allocation (on GPU) and data transfers between CPU and GPU.

3.2 Data Distribution on Multiple GPUs

In applications working on container data types (vectors, matrices, etc.) GPU's often access disjoint parts of input data, such that copying only a part of the container to a GPU would be more efficient than copying the whole data to each GPU. To simplify the specification of partitionings of containers in programs

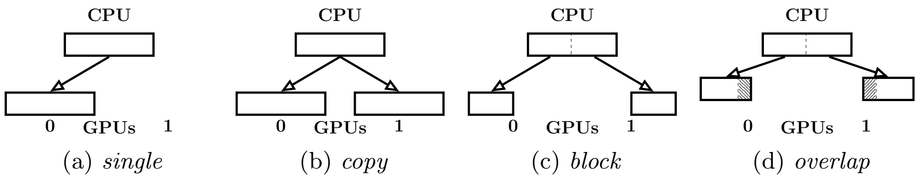


Fig. 1. Distributions of a vector in SkelCL

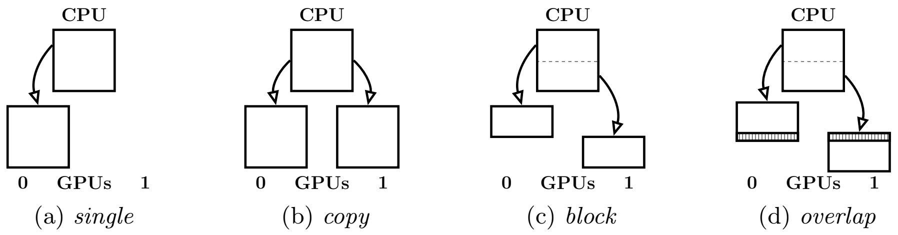


Fig. 2. Distributions of a matrix in SkelCL

for multi-GPU systems, SkelCL implements the *distribution* mechanism that describes how a container is distributed among the available GPUs. It allows the programmer to abstract from managing memory ranges which are shared or spread across multiple GPUs: the programmer can think of a distributed container as of a self-contained entity.

Four kinds of distribution are currently available in SkelCL: *single*, *copy*, *block*, and *overlap* (see Fig. 1 for distributing a vector on a system with two GPUs). If distribution is set to *single* (Fig. 1a), than vector's whole data is stored on a single GPU (the first GPU if not specified otherwise). The *copy* distribution (Fig. 1b) copies vector's entire data to each available GPU. With the *block* distribution (Fig. 1c), each GPU stores a contiguous, disjoint chunk of the vector. The *overlap* distribution (Fig. 1d) stores on each GPU the chunk like in the block distribution, together with one or several border elements of the neighboring chunk.

The same four distributions are provided also for the matrix data type (Figure 2). In particular the overlap distribution splits the matrix into one chunk for each GPU; in addition, each chunk contains a number of continuous rows from the neighboring chunks. A parameter – the *overlap size* – specifies the number of rows at the borders of a chunk which are copied to the two neighboring GPUs. Figure 2d illustrates the overlap distribution: GPU 0 receives the top chunk ranging from the top row to the middle, while GPU 1 receives the second chunk ranging from the middle row to the bottom. The marked parts are called *overlap region* they are the same on both GPUs.

The application developer can set the distribution of containers (vectors and matrices) explicitly, otherwise every skeleton selects a default distribution for its input and output containers. Container's distribution can be changed at runtime: this implies data exchanges between multiple GPUs and the CPU, which are performed by the SkelCL implementation implicitly. Implementing such data transfers in the standard OpenCL is a cumbersome task: data has to be downloaded to the CPU before it is uploaded to the GPUs, including the corresponding length and offset calculations; this results in a lot of low-level code which becomes completely hidden when using SkelCL.

3.3 Basic Patterns of Parallelism (Skeletons)

In original OpenCL, computations are expressed as *kernels* which are executed in a parallel manner on a GPU: the application developer must explicitly specify how many instances of a kernel are launched. In addition, kernels usually take pointers to GPU memory as input and contain program code for reading/writing single data items from/to it. These pointers have to be used carefully, because no boundary checks are performed by OpenCL.

To shield the application developer from these low-level programming issues, SkelCL extends OpenCL by introducing high-level programming patterns, called *algorithmic skeletons* [15]. Formally, a skeleton is a higher-order function that executes one or more user-defined (so-called *customizing*) functions in a pre-defined parallel manner, while hiding the details of parallelism and communication from the user [15].

The current version of SkelCL provides six skeletons: *map*, *zip*, *reduce*, *scan*, *mapOverlap* and *allpairs*. We define first the four basic skeletons. We do this semi-formally, with c, cl and cr denoting vectors with elements c_i, cl_i and cr_i where $0 < i \leq n$:

- The map skeleton applies a unary customizing function f to each element of an input vector c, i. e.

  $$map\ f\ [c_1, c_2, \ldots, c_n] = [f(c_1), f(c_2), \ldots, f(c_n)]$$

  In a SkelCL program, a map skeleton is created as an object for a unary function f, e. g. negation, like this:

  ```
  Map<float(float)> neg("float func(float x){ return -x;}");
  ```

  This map object can then be called as a function with a vector as argument:

  ```
  resultVector = neg( inputVector );
  ```

- The zip skeleton operates on two vectors cl and cr, applying a binary customizing operator $\oplus$ pairwise:

  $$zip\ (\oplus)\ [cl_1, cl_2, \ldots, cl_n]\ [cr_1, cr_2, \ldots, cr_n] = [cl_1 \oplus cr_1, cl_2 \oplus cr_2, \ldots, cl_n \oplus cr_n]$$

  In SkelCL, a zip skeleton object for adding two vectors is created like as:

```
Zip<float(float, float)> add("float func(float x,float y){return x+y;}");
```

  and can then be called as a function with a pair of vectors as arguments:

  ```
  resultVector = add( leftVector, rightVector );
  ```

- The reduce skeleton computes a scalar value from a vector using a binary associative operator $\oplus$, i. e.

$$red\ (\oplus)\ [v_1, v_2, \ldots, v_n] = v_1 \oplus v_2 \oplus \cdots \oplus v_n$$

For example, to sum up all elements of a vector, the reduce skeleton is created with addition as customizing function, and called as follows:

```
Reduce<float(float)> sumUp("float func(float x,float y){ return x+y;}");
                   result = sumUp( inputVector );
```

– The scan skeleton (a. k. a. prefix-sum) yields an output vector with each element obtained by applying a binary associative operator $\oplus$ to the elements of the input vector up to the current element's index, i. e.

$$scan\ (\oplus)\ [v_1, v_2, \ldots, v_n] = [v_1, v_1 \oplus v_2, \ldots, v_1 \oplus v_2 \oplus \cdots \oplus v_n]$$

The prefix sums customized with addition is specified and called in SkelCL as follows:

```
Scan<float(float)> prefixSum("float func(float x,float y){return x+y;}");
                 result = prefixSum( inputVector );
```

In SkelCL, rather than writing low-level kernels, the application developer customizes suitable skeletons by providing application-specific functions which are often much simpler than kernels as they specify an operation on basic data items rather than containers. Skeletons can be executed on both single- and multi-GPU systems. In case of a multi-GPU system, the calculation specified by a skeleton is performed automatically on all GPUs available in the system.

```
int main (int argc, char const* argv[]) {
  SkelCL::init(); /* initialize SkelCL */
/* create skeletons */
  SkelCL::Reduce<float> sum ("float sum (float x,float y)\
      {return x+y;}");
  SkelCL::Zip<float>    mult("float mult(float x,float y)\
      {return x*y;}");
/* create input vectors */
  SkelCL::Vector<float> A(SIZE);
  SkelCL::Vector<float> B(SIZE);
/* fill vectors with data */
  fillVector(A.begin(), A.end());
  fillVector(B.begin(), B.end());
/* execute skeleton */
  SkelCL::Scalar<float> C = sum( mult( A, B ) );
/* fetch result */
  float c = C.getValue();
}
```

Listing 1.1. SkelCL program computing the dot product of two vectors. Arrays a_ptr and b_ptr initialize the vectors.

Listing 1.1 shows how a dot product of two vectors is implemented in SkelCL using two of the basic skeletons. Here, the `Zip` skeleton is customized by multiplication, and the `Reduce` skeleton is customized by usual addition. For comparison, an OpenCL-based implementation of the dot product computation provided by NVIDIA requires approximately 68 lines of code (kernel function: 9 lines, host program: 59 lines) [2].

3.4 The MapOverlap Skeleton

Many numerical and image processing applications dealing with two-dimensional data perform calculations for a particular data element (e. g., a pixel) taking neighboring data elements into account. To facilitate the development of such applications, we define in SkelCL a skeleton that can be used with both vector and matrix data type; we explain the details for the matrix data type.

– The *MapOverlap* skeleton takes two parameters: a unary function f and an integer value d. It applies f to each element of an input matrix m_{in} while taking the neighboring elements within the range $[-d, +d]$ in each dimension into account, i. e.

$$m_{out}[i,j] = f\begin{pmatrix} m_{in}[i-d,j-d] \dots m_{in}[i-d,j] \dots m_{in}[i-d,j+d] \\ \vdots & \vdots & \vdots \\ m_{in}[i,j-d] \quad \dots \quad m_{in}[i,j] \quad \dots \quad m_{in}[i,j+d] \\ \vdots & \vdots & \vdots \\ m_{in}[i+d,j-d] \dots m_{in}[i+d,j] \dots m_{in}[i+d,j+d] \end{pmatrix}$$

In the actual source code, the application developer provides the function f which receives a pointer to the element in the middle, $m_{in}[i, j]$.

Listing 1.2 shows a simple example of computing the sum of all direct neighboring values using the MapOverlap skeleton. To access the elements of the input matrix m_{in}, function `get` is provided by SkelCL. All indices are specified relative to the middle element $m_{in}[i, j]$; therefore, for accessing this element the function call `get(m_in, 0, 0)` is used. The application developer must ensure that only elements in the range specified by the second argument d of the MapOverlap skeleton, are accessed. In Listing 1.2, range is specified as $d = 1$, therefore, only direct neighboring elements are accessed. To enforce this property, boundary

```
MapOverlap<float(float)> m("float func(float* m_in){
float sum = 0.0f;
for (int i = -1; i < 1; ++i)
    for (int j = -1; j < 1; ++i)
        sum += get(m_in, i, j); return sum;
}", 1, SCL_NEUTRAL, 0.0f);
```

Listing 1.2. MapOverlap skeleton computing the sum of all direct neightbors for every element in a matrix

```
__kernel void sum_up(__global float* m_in,
                     __global float* m_out,
                     int width, int height) {
  int i_off = get_global_id(0);
  int j_off = get_global_id(1);
  float sum = 0.0f;
  for (int i = i_off - 1; i < i_off + 1; ++i)
    for (int j = j_off - 1; j < j_off + 1; ++j) {
      // perform boundary checks
      if ( i < 0 || i > width || j < 0 || j > height )
        continue;
      sum += m_in[ j * width + i ];     }
  m_out[ j_off * width + i_off ] = sum; }
```

Listing 1.3. An OpenCL kernel performing the same calculation as the MapOverlap skeleton shown in Listing 1.2

checks are performed at runtime by the `get` function. In future work, we plan to avoid boundary checks at runtime by statically proving that all memory accesses are in bounds, as it is the case in the shown example.

Special handling is necessary when accessing elements out of the boundaries of the matrix, e.g., when the item in the top-left corner of the matrix accesses elements above and left of it. The MapOverlap skeleton can be configured to handle such out-of-bound memory accesses in two possible ways: 1) a specified neutral value is returned; 2) the nearest valid value inside the matrix is returned. In Listing 1.2, the first option is chosen and 0.0 is provided as neutral value.

Listing 1.3 shows how the same simple calculation can be performed in standard OpenCL. While the amount of lines of code increases by a factor of 2, the complexity of each single line also increases: 1) Besides a pointer to the output memory, the width of the matrix has to be provided as parameter; 2) the correct index has to be calculated for every memory access using an offset and the width of the matrix, i. e. knowledge about how the two-dimensional matrix is stored in one-dimensional memory is required. 3) In addition, manual boundary checks have to be performed to avoid faulty memory accesses.

SkelCL avoids all these low-level details. Neither additional parameter, nor index calculations or manual boundary checks are necessary.

3.5 The Allpairs Skeleton

All-pairs computations occur in a variety of applications, ranging from pairwise Manhattan distance computations used in bioinformatics [12] to N-Body simulations used in physics [3]. All these applications follow a common computational scheme: for two sets of entities, the same computation is performed independently for all pairs of entities from the first set combined with entities from the second set. An entity is usually described by a d-dimensional vector.

We define the all-pairs computation scheme for two sets of n and m entities, each entity represented by a d-dimensional vector. We represent the sets as an

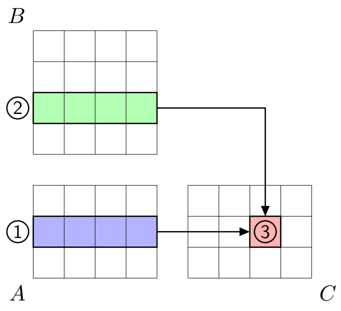


Fig. 3. The allpairs computation: Element $c_{2,3}$ (③) is computed by combining the second row of A (①) with the third row of B (②) using the binary operator $\oplus$

$n \times d$ matrix A and an $m \times d$ matrix B. The all-pairs computation yields an output matrix C of size $n \times m$ as follows: $C_{i,j} = A_i \oplus B_j$, where A_i and B_j are rows of A and B, correspondingly: $A_i = [A_{i,1}, \cdots, A_{i,d}]$, $B_j = [B_{j,1}, \cdots, B_{j,d}]$, and $\oplus$ is a binary function applied to every pair of rows from A and B.

Figure 3 illustrates this definition: the element marked as ③ of matrix C is computed by combining the second row of A marked as ① with the third row of B marked as ② using the binary operator $\oplus$.

For formally defining the all-pairs skeleton, let d, m and n be positive numbers. Let A be a $n \times d$ matrix, B be a $m \times d$ matrix and C be a $n \times m$ matrix with their entries $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$ respectively. Let $\oplus$ be a binary function on vectors. The algorithmic skeleton *allpairs* is defined as follows:

$$allpairs(\oplus)\left(\begin{bmatrix} a_{1,1} & \cdots & a_{1,d} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,d} \end{bmatrix}, \begin{bmatrix} b_{1,1} & \cdots & b_{1,d} \\ \vdots & & \vdots \\ b_{m,1} & \cdots & b_{m,d} \end{bmatrix}\right) \stackrel{\text{def}}{=} \begin{bmatrix} c_{1,1} & \cdots & c_{1,m} \\ \vdots & & \vdots \\ c_{n,1} & \cdots & c_{n,m} \end{bmatrix}$$

with entries $c_{i,j}$ of the computed $n \times m$ matrix C defined as:

$$c_{i,j} = [a_{i,1} \ \cdots \ a_{i,d}] \oplus [b_{j,1} \ \cdots \ b_{j,d}]$$

To illustrate the definition, we show how matrix multiplication can be expressed using the allpairs skeleton.

Example 1: The matrix multiplication is a basic linear algebra operation, which is a building block of many scientific applications. A $n \times d$ matrix A is multiplied by a $d \times m$ matrix B, producing a $n \times m$ matrix $C = A \times B$ whose element $C_{i,j}$ is computed as the dot product of the ith row of A with jth column of B. The dot product of two vectors a and b of length d is computed as:

$$dotProduct(a, b) = \sum_{k=1}^{d} a_k \cdot b_k \tag{1}$$

The matrix multiplication can be expressed using the allpairs skeleton as:

$$A \times B = allpairs(dotProduct)\left(A, B^T\right) \tag{2}$$

where B^T is the transpose of matrix B.

4 Application Studies and Experiments

We consider two application case studies using the SkelCL library: 1) the calculation of a Mandelbrot fractal, and 2) the Sobel edge detection. Both SkelCL implementations are compared to similar implementations in CUDA and OpenCL regarding their programming effort and runtime performance.

For our runtime experiments we use a PC with a quad-core CPU (Intel Xeon E5520, 2.26 GHz) and 12 GB of memory. The system is connected to a Tesla S1070 computing system equipped with 4 Tesla GPUs. Its dedicated 16 GB of memory (4 GB per GPU) is accessed with up to 408 GB/s (102 GB/s per GPU). Each GPU comprises 240 streaming processor cores running at 1.44 GHz.

4.1 Application Study: Mandelbrot Set

The Mandelbrot set calculation [10] is a time-consuming task which is often used as a benchmark. Computing a Mandelbrot fractal is easily parallelizable, as all pixels can be computed simultaneously. As the criteria for programming effort we use the number of Lines of Code (LoC), the results are in Fig. 4.

We created three similar parallel implementations for computing a Mandelbrot fractal using CUDA, OpenCL, and SkeCL.

CUDA and SkelCL require a single line of code for initialization in the host code, whereas OpenCL requires a lengthy creation and initialization of different

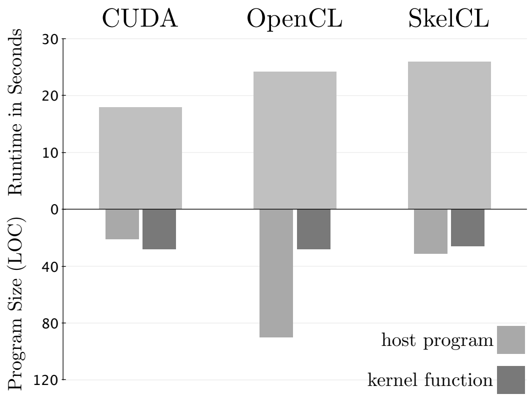


Fig. 4. Runtime and program size of the Mandelbrot application

data structures which take about 20 LoC. The host CPU code differs significantly between all implementations. In CUDA, the kernel is called like an ordinary function. A proprietary syntax is used to specify the size of work-groups executing the kernel. In OpenCL, several API functions are called to load and build the kernel, pass arguments to it and launch it using a specified work-group size. In SkelCL, the kernel is passed to a newly created instance of the `Map` skeleton. A `Vector` of complex numbers, each of which represents a pixel of the Mandelbrot fractal, is passed to the `Map` skeleton upon execution. Specifying the work-group size is mandatory in CUDA and OpenCL, whereas this is optional in SkelCL.

The OpenCL-based implementation has in total 118 lines of code (kernel: 28 lines, host program: 90 lines) and is thus more than twice as long as the CUDA and SkelCL versions with 49 lines (28, 21) and 57 lines (26, 31), respectively (see Figure 4).

We tested our implementations on a single GPU of our test system to compute a Mandelbrot fractal of size 4096×3072 pixels. In CUDA and OpenCL, work-groups of 16×16 are used; SkelCL uses its default work-group size of 256.

As compared to the runtime of the SkelCL-based implementation (26 sec), the implementation based on OpenCL (25 sec) and CUDA (18 sec) are faster by 4% or 31%, respectively. Since SkelCL is built on top of OpenCL, the performance difference of SkelCL and OpenCL can be regarded as the overhead introduced by SkelCL. Previous work [9] reported that CUDA was usually faster than OpenCL, which explains the higher performance of the implementation based on CUDA. The Mandelbrot application demonstrates that SkelCL introduces a tolerable overhead of less than 5% as compared to OpenCL.

4.2 Application Study: Sobel Edge Detection

To evaluate the usability and performance of the MapOverlap skeleton on the matrix data type, we implemented the Sobel edge detection that produces an output image in which the detected edges in the input image are marked in white and plain areas are shown in black.

```
for (i = 0; i < width; ++i)
  for (j = 0; j < height; ++j)
    h = -1*img[i-1][j-1] +1*img[i+1][j-1]
        -2*img[i-1][j  ] +2*img[i+1][j  ]
        -1*img[i-1][j+1] +1*img[i+1][j+1];
    v = ...;
    out_img[i][j] = sqrt(h*h + v*v);
```

Listing 1.4. Sequential implementation of the Sobel edge detection

Listing 1.4 shows the algorithm of the Sobel edge detection in pseudo-code, with omitted boundary checks for brevity. In this sequential version, for computing an output value `out_img[i][j]` the input value `img[i][j]` and the direct neighboring elements are needed. Therefore, the MapOverlap skeleton is a perfect fit for implementing the Sobel edge detection.

```
// skeleton customized with Sobel edge detection algorithm
MapOverlap<char(char)> m( "char func(const char* img) {
  short h = -1*get(img,-1,-1) +1*get(img,+1,-1)
            -2*get(img,-1, 0) +2*get(img,+1, 0)
            -1*get(img,-1,+1) +1*get(img,+1,+1);
  short v = ...;
  return sqrt(h*h + v*v); }", 1, SCL_NEUTRAL, 0);
Matrix<char> out_img = m(img); // execution of the skeleton
```

Listing 1.5. SkelCL implementation of the Sobel edge detection

Listing 1.5 shows the SkelCL implementation using the MapOverlap skeleton and the matrix data type. The implementation is straightforward and very similar to the sequential version in Listing 1.4. The only notable difference is that for accessing elements the `get` function is used instead of the square bracket notation.

```
__kernel void sobel_kernel( __global const uchar* img,
                            __global       uchar* out_img)
 uint i = get_global_id(0);   uint j = get_global_id(1);
 uint w = get_global_size(0); uint h = get_global_size(1);
 // perform boundary checks
 if(i >= 1 && i < (w-1) && j >= 1 && j < (h-1)) {
  char ul = img[((j-1)*w)+(i-1)];
  char um = img[((j-1)*w)+(i+0)];
  char ur = img[((j-1)*w)+(i+1)];
  // ... 5 more
  char lr = img[((j+1)*w)+(i+1)];

  out_img[j * w + i] = computeSobel(ul, um, ur, ..., lr); } }
```

Listing 1.6. Additional boundary checks and index calculations for Sobel algorithm, necessary in the standard OpenCL implementation

Listing 1.6 shows a part of the OpenCL implementation for Sobel edge detection provided by AMD as an example for their software development kit [1]. The actual computation is performed inside the `computeSobel` function, which is omitted in the listing, since it is quite similar to the sequential version in Listing 1.4. The listing shows that extra low-level code is necessary to deal with technical details, like boundary checks and index calculations, which are arguably complex and error-prone.

We performed runtime experiments using one NVIDIA Tesla GPU with 480 processing elements and 4 GByte memory. Figure 5 shows the runtime of two OpenCL versions (from AMD and NVIDIA SDK) vs. the SkelCL version with the MapOverlap skeleton presented in Listing 1.5. Only the kernel runtimes are shown, as the data transfer times are equal for all versions. Measurements were taken using the OpenCL profiling API. We used the popular Lena image [18] with a size of 512×512 pixel and took the mean values of six runs. The AMD version

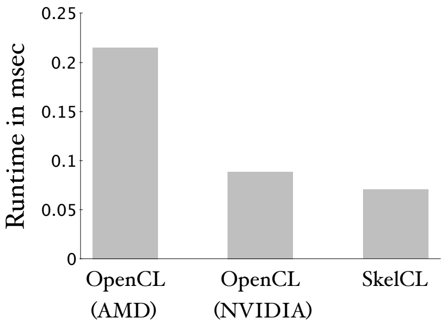


Fig. 5. Performance results for Sobel edge detection

is clearly slower then the two other implementations, because it does not use the fast local memory which the NVIDIA implementation and the MapOverlap skeleton of SkelCL do. SkelCL totally hides the memory management details inside its implementation from the application developer. The NVIDIA and SkelCL implementations perform similar. In this particular example, SkelCL even slightly outperforms the implementation by NVIDIA.

In addition to the performance advantage over the AMD and NVIDIA versions, the SkelCL program is also significantly simpler than the cumbersome OpenCL implementation. The SkelCL program only comprises the few lines of code shown in Listing 1.5. The AMD implementations requires 37 lines of code for its kernel implementation and the NVIDIA implementation requires even 208 lines of code. Both versions require additional lines of code for the host program which manages the execution of the OpenCL kernel. No index calculations or boundary checks are necessary in the SkelCL version whereas they are crucial for a correct implementation in OpenCL.

5 Conclusion and Related Work

This paper presents the SkelCL high-level programming model for multi-GPU systems and its implementation as a library. The SkelCL programming model significantly raises the level of abstraction: it combines parallel patterns to express computations, parallel container data types for simplified memory management and a data (re)distribution mechanism to improve scalability in systems with multiple GPUs. The SkelCL library is available as open source software from `http://skelcl.uni-muenster.de`.

There are a number of other projects aiming at high-level GPU programming. *SkePU* [17] provides a vector class similar to our `Vector` class, but unlike SkelCL it does not support different kinds of data distribution on multi-GPU systems. SkelCL provides a more flexible memory management than SkePU, as data transfers can be expressed by changing data distribution settings. Both approaches differ significantly in the way how functions are passed to skeletons.

While functions are defined as plain strings in SkelCL, SkePU uses a macro language, which brings some serious drawbacks. For example, it is not possible to call mathematical functions like sin or cos inside a function generated by a SkelPU macro, because these functions are either named differently in all of their three target programming models (CUDA, OpenCL, OpenMP) or might even be missing entirely. The same holds for functions and keywords related to performance tuning, e. g., the use of local memory. SkelCL does not suffer from these drawbacks because it relies on OpenCL and thus can be executed on a variety of GPUs and other accelerators.

CUDPP [4] provides data-parallel algorithm primitives similar to skeletons. These primitives can be configured using only a predefined set of operations, whereas skeletons in SkelCL are true higher-order functions which accept any user-defined function. CUDPP does not simplify data management, because data still has to be exchanged between CPU and GPU explicitly. There is also no support for multi-GPU applications.

Thrust [16] provides two vector types similar to the vector type of the C++ Standard Template Library. While these types refer to vectors stored in CPU or GPU memory, respectively, SkelCL's vector data type provides a unified abstraction for CPU and GPU memory. Thrust also contains data-parallel implementations of higher-order functions, similiar to SkelCL's skeletons. SkelCL adopts several of Thrust's ideas, but it is not limited to CUDA-capable GPUs and supports multiple GPUs.

Unlike SkelCL, *OpenACC* [6], *PGI Acccelerator* [5], and *HMPP* [8] are compiler-based approaches to GPU programming, similar to the popular OpenMP [7]. The programmer uses compiler directives to mark regions of code to be executed on a GPU. A compiler generates executable code for the GPU, based on the used directives. Although source code for low-level details like memory allocation or data exchange is generated by the compiler, these operations still have to be specified explicitly by the programmer using suitable compiler directives. We consider these approaches low-level, as they do not perform data transfer automatically to shield the programmer from low-level details and parallelism is still expressed explicitly.

Acknowledgments. This work is partially supported by the OFERTIE (FP7) and MONICA projects. We would like to thank NVIDIA for their generous hardware donation.

References

1. AMD APP SDK code samples, version 2.7 (February 2013), `http://developer.amd.com/`
2. NVIDIA CUDA SDK code samples, version 5.0 (February 2013), `http://developer.nvidia.com/`
3. Arora, N., Shringarpure, A., Vuduc, R.W.: Direct N-body Kernels for Multicore Platforms. In: Proceedings of the 2009 International Conference on Parallel Processing, ICPP 2009, pp. 379–387. IEEE Computer Society, Washington, DC (2009)

4. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Graphics Hardware 2007 (2007)
5. T.P. Group: PGI Accelerator Programming Model for Fortran & C (2010)
6. OpenACC Application Program Interface. version 1.0 (2011), `http://www.openacc.org/`
7. OpenMP Application Program Interface. OpenMP Architecture Review Board, version 3.0 (2008), `http://www.openmp.org/mp-documents/spec30.pdf`
8. Bihan, S., Moulard, G., Dolbeau, R., et al.: Directive-based heterogeneous programming a GPU-accelerated RTM use case. In: Proceedings of the 7th International Conference on Computing, Communications and Control Technologies (2009)
9. Kong, J., Dimitrov, M., Yang, Y., et al.: Accelerating MATLAB image processing toolbox functions on GPUs. In: GPGPU 2010: Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. ACM (2010)
10. Mandelbrot, B.B.: Fractal aspects of the iteration of $z \mapsto \lambda z(1-z)$ for complex λ and z. Annals of the New York Academy of Sciences 357, 249–259 (1980)
11. NVIDIA CUDA API Reference Manual, version 5.0 (February 2013)
12. Chang, D., Desoky, A., Ouyang, M., Rouchka, E.: Compute Pairwise Manhattan Distance and Pearson Correlation Coefficient of Data Points with GPU. In: Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, SNPD 2009, pp. 501–506 (2009)
13. Munshi, A.: The OpenCL Specification, version 1.2
14. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL – A Portable Skeleton Library for High-Level GPU Programming. In: 2011 IEEE 25th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 1171–1177 (2011)
15. Gorlatch, S., Cole, M.: Parallel skeletons. In: Encyclopedia of Parallel Computing, pp. 1417–1422 (2011)
16. Hoberock, J., Bell, N.: Thrust: A Parallel Template Library (2009)
17. Enmyren, J., Kessler, C.: SkePU: A multi-backend skeleton programming library for multi-GPU systems. In: Proceedings 4th Int. Workshop on High-Level Parallel Programming and Applications, pp. 5–14 (2010)
18. University of Southern California SIPI Image Database. Girl (lena, or lenna), `http://sipi.usc.edu/database/database.php?volume=misc`

Cellular Automata Model of Some Organisms Population in Lake Baikal

Ivan Afanasyev

Russian Academy of Sciences,
Mathematical Department,
Novosibirsk Scientific Centre,
Institute of Computational Mathematics and Mathematical Geophysics SB RAS
ivafanas@gmail.com

Abstract. A cellular automata model of population dynamics of eight organisms in Lake Baikal is proposed and investigated. The model allows to take into account spatial organisms distribution, seasonal dependency of birth rates, possible habitat pollution and water streams. Computational experiment is presented. It demonstrates that population dynamics tends to stable oscillating process with period equal to 1 year. The model was verified within production-to-biomass and frequency of occurrence ratios.

Keywords: cellular automata, discrete model, population dynamics, Lake Baikal.

1 Introduction

Cellular Automata (CA) models are an approach for investigation of self-organization processes. It allows simulating complex nonlinear processes using comparatively simple rules. The research of CA-models for self-organization processes has been done by Wolfram [1], Chua [2].

More complex processes require usage of parallel composition of CA which was shown in [3] as an example of abstract prey-predator system.

Here a CA-model is proposed, that allows to simulate spatial distribution of organisms when some perturbation occur. This model takes into account eight organisms groups, time and spatial dependency of environment influence, seasonal dependency of birth rates, possible habitat pollution and water streams.

2 Task Statement

Comephorus is the most wide-spread fish in the Lake. Its food is macrohectopus and its own youngest. Following [4] three species are investigated: macrohectopus, comephorus dybovksi and comephorus baikalensis. Each species is divided onto age groups:

Macrohectopus: m_1 - immature, m_2 – puberal;
Comephorus dybovksi: d_1 - one-year-old, d_2 - immature, d_3 – puberal;
Comephorus baikalensis: b_1 - one-year-old, b_2 - immature, b_3 – puberal.

Groups m_1, b_1, d_1 are preys. Groups b_2, b_3, d_2, d_3 are predators. All predators eat all preys with different preferences rates. Younger individuals grow up to elder age groups. The oldest individuals propagate.

3 Composite CA-Model

Let us define a CA-model of the population dynamic of Lake Baikal organisms as

$$\mathrm{N} = \langle \Sigma, M, F, \rho \rangle \tag{1}$$

Σ is a cell states set, M is a cell naming set, F is a global operator, ρ is a functional mode. M is split into 8 pair wise disjoint subsets M_α^i, one per group of organisms. Each subset can be presented by a square mesh Q.

$$M = M_m^1 \cup M_m^2 \cup M_d^1 \cup M_d^2 \cup M_d^3 \cup M_b^1 \cup M_b^2 \cup M_b^3 \tag{2}$$

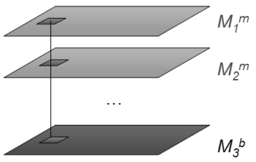


Fig. 1. Subsets M_α^i

A c*ell* is an element of $M \times \Sigma$. Cells states are integers, representing model density of organisms in this cell. Cells $\langle m_1,n_1\rangle \in M_\alpha^i$ and $\langle m_2,n_2\rangle \in M_\beta^j$ are called *twins* if m_1 and m_2 correspond to the same cell of Q.

$S(m) = \langle (\varphi_1(m), n_1), \ldots, (\varphi_k(m), n_k) \rangle$ is a *local configuration*, $n_i \in \Sigma$, $\varphi_i : M \rightarrow M$. Let us define *local operator f* as:

$$f : \{S(m)\} \rightarrow \{S(m)\} \tag{3}$$

Application of *f* to a cell $\langle m, n\rangle$ is a replacement of $S(m)$ by $f(S(m))$. *Iteration* or *global operator application* is the application of local operator to all cells.

In the model two global operators are used: F_1 is for motion and F_2 is for prey-predator interactions. Two basic functional modes are known: *synchronous* and *asynchronous* [3]. The proposed CA-model uses more complex mode: F_1 is applied asynchronously several times to each M_α^i independently, F_2 is applied synchronously.

3.1 Motion Operator

F_1 is a sequential superposition of diffusion operator F_d and streams operator F_s.

$$F_1 = F_d \circ F_s \tag{4}$$

Let F_h be the integer diffusion operator given in [5]. F_h is applied to each set M_α^i separately. Since speeds are different for different species, F_h is applied K^α_i times per one iteration to each set M_α^i. Ratios K^α_i depend on cruising speeds of individuals.

$$F_d\Big|_{M_\alpha^i} = \left(F_h\right)^{K_\alpha^i} \tag{5}$$

Operator F_s is obtained according to a function *stream*: $M \rightarrow R^2$ which determines water stream speed and direction. Application of F_s is equal to moving of individuals from cell $\langle m, n\rangle$ in direction *stream(m).*

3.2 Density Changing Operator

The local density changing operator f_2:

$$f_2 : \{S_2(m)\} \rightarrow \{S_2(m)\} \tag{6}$$

where $S_2(m)$ is the twin cells. f_2 is applied synchronously to all cells of M. The new cell state after applying f_2 is:

$$n_i^{\alpha}{}' = n_i^{\alpha} + \left(\rho_i^{\alpha} n_j^{\alpha} - \lambda_i^{\alpha} n_i^{\alpha} - \theta_i^{\alpha} n_i^{\alpha}\right)\Delta t \tag{7}$$

where j is the age group number, whose individuals produce the individuals of i-th age group; $\rho^{\alpha}{}_i n_j$ is the growth of quantity of i-th age group due to propagation (case i=1) or aging; $\lambda^{\alpha}{}_i n_i$ is the number of dead individuals; $\theta^{\alpha}{}_i\, n_i$ is the number of grown up individuals; Δt is the physical time step which corresponds to one model iteration.

To take into account seasonal birthrates, the values of $\rho^d{}_1$ and $\rho^b{}_1$ are multiplied by the periodic functions $season_d(t)$ and $season_b(t)$ whose period equals to one year. The possible habitat pollution is taken into account. Let $poll(m)$: $M \rightarrow R_+$ be a pollution map. Death rates $\lambda^{\alpha}{}_i$ given in [4] are updated to be as follows:

$$\lambda_i^{\alpha}{}' = \left(1 + poll(m)\right)\lambda_i^{\alpha} \tag{8}$$

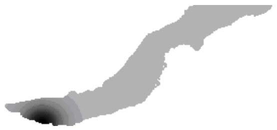


Fig. 2. Pollution map *poll(m)*. Darker color means bigger pollution intensity.

Pollution map used in experiment is presented in Fig 2. It is a density of two-dimensional standard normal distribution with center in cell m_0 in the south part of the Lake Baikal.

4 Computational Experiment

Initial state is uniform distribution of individuals over the all model area according to the stable state taken from [4]. Since pollution influence is not hardly investigated, $poll(m_0)$ is taken equal to 7, which means the 8 times growth of death rates in cell m_0.

States for southern area of Lake Baikal for impuberal macrohectopus (m_1) and puberal comephorus baikalensis (b_3) on 2000-th iteration are shown in Fig 3.

Pollution influence does not spread out of the polluted area significantly.

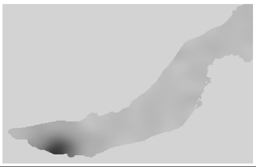
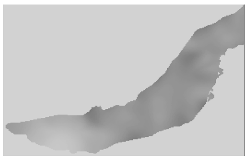
	
impuberal macrohectopus (m_1)	puberal comephorus baikalensis (b_3)

Fig. 3. States for m_1 and b_3 on 2000-th iteration. Darker color means higher density.

Population dynamics is shown in Fig. 4. In the polluted area the density of macrohectopus is significantly increased due to reduction of predatory groups.

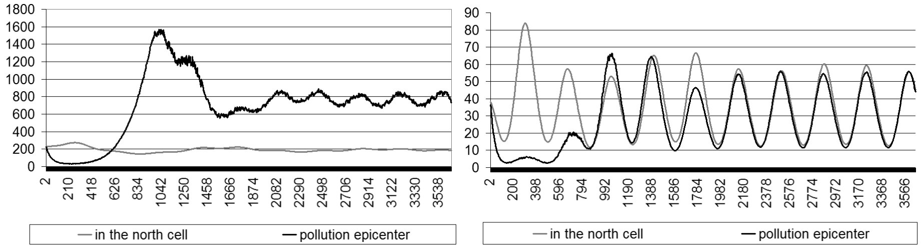


Fig. 4. Populations' dynamics in pollution epicenter and in the north of Lake Baikal. Left image presents comparison of dynamics of macrohectopus. Right image presents comparison of dynamics of comephorus dubovsky.

5 Model Verification

Verification was done for the non-polluted case. The following criteria were used:

- P_α / B_α – production-to-biomass ratio for each species $\alpha \in \{m, d, b\}$,
- N_α / N_β – density-to-density ratio between different species $\alpha, \beta \in \{m, d, b\}$, N being the average annual density of organisms.

Model verification results are shown in Table 1. Modeling results differ from assessments given in [4, 6, 7] in about 20%.

Table 1. Model verification results

	P_m / B_m	$(P_d + P_b) / (B_d + B_b)$	N_m / N_d	N_m / N_b
in [4, 6, 7]	3 - 8	1,24	6,05	21,52
model	5,77	1,49	6,00	20,45

6 Conclusions

CA-model of organism population dynamics is proposed. It allows to take into account spatial individuals distribution, seasonal dependency of birth rates, possible habitat pollution and water streams.

Computational experiment is carried out. Population dynamics tends to stable oscillating process with period equals to 1 year. More intensive pollution means more significant changes in population dynamics and final stable oscillation process, but the pollution influence does not spread out of the polluted area.

Model verification was done for the non-polluted case. Verification results within production-to-biomass and occurrence frequency criteria differ from assessments in [4, 6, 7] in about 20%.

References

1. Wolfram, S.: A new kind of science, 1197 p. Wolfram Media Inc., USA (2002)
2. Chua, L.O.: CNN: A Paradigm for Complexity, 320 p. World Scientific Series on Nonlinear Science. University of California, Berkely (1998)
3. Bandman, O.L.: A method for construction of cellular automata simulating pattern formation processes. Theoretical Background of Applied Discrete Mathematics (4), 91–99 (2010)
4. Zorkalcev, V.I., Kazazaeva, A.V., Mokry, I.V.: Model of three pelagic organisms of Lake Baikal. Modern Technologies. System Analysis. Modeling (1), 182–193 (2008) (in Russian)
5. Medvedev, Y.: Multi-particle Cellular Automata Models For Diffusion Simulation. In: Hsu, C.-H., Malyshkin, V. (eds.) MTPP 2010. LNCS, vol. 6083, pp. 204–211. Springer, Heidelberg (2010)
6. Mazepova, G.F., Timoshkin, O.A., Melnik, N.G., Obolkina, L.A.: Guide and key to pelagic animals of Baikal with ecological notes. Nauka, Novosibirsk, 693 p. (1995) (in Russian)
7. Starikov, G.V.: Comephorus. Nauka, Novosibirsk, 94 p. (1977) (in Russian)

3-D Cellular Automata Model of Fluid Permeation through Porous Material[⋆]

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 3D Cellular Automata (CA) model for simulating fluid permeation through porous material with complex morphology is developed and investigated. The model is a composition of two interacting CA: one, simulating fluid convection, induced either by gravitation force or by external pressure, and another — simulating fluid surface leveling by diffusion. Both CA process the same discrete space, their operation being separated in time and space, which simplifies essentially parallel implementation. The CA model is tested on an example of water permeation through soil. Results of its parallel implementation on a multiprocessor with distributed memory are presented. A tomographic digitized representation of a 3D soil sample was kindly given by Prof.Wim Cornelis.[1] The simulation program was implemented on the cluster of Siberian Supercomputer Center of Siberian Branch of Russian Academy of Science.

1 Introduction

In connection with new technologies of production and implementation of porous materials new claims are set to computer simulation of processes in them. Dealing with porous media in the same manner that with bulk materials, does not satisfy neither designers of new composites, nor researchers of soil fertility. In order to understand how the porous material interacts with the permeating fluid, it is necessary to make allowance for internal properties of the material, among which morphology and interaction character between pore walls and incoming fluid are the most important. Unfortunately, conventional mathematics, based on partial differential equations, cannot be helpful, because of impossibility to describe pore borders by means of continuous functions [1,2]. This is why, several attempts are made to develop discrete methods of fluid permeation through porous media, because they "do not fear intricate boundary conditions".

⋆ Supported by (1)Presidium of Russian Academy of Sciences, Basic Research Program 15-9 (2013), (2) Siberian Branch of Russian Academy of Sciences, Interdisciplinary Project 47, (3) Russian Fund for Basic Research grant 11-01-00567a.

[1] Prof. Wim Cornelis, SoPHy, Department Soil Management, Ghent University, Coupure links 653, 9000 Gent, Belgium.

Such methods are based on Lattice-Gas [3] and Lattice-Boltzmann [4] discrete hydrodynamics, conventional cellular automata [5,6] being also considered. Among them, the models based on Lattice-Gas principles are predestined to simulate laminar flows. They are appropriate when the flow constantly goes through a piece of porous material, as it happens in polymer membranes or in carbon electrodes of hydrogen energetic cells [1]. But if transient processes are of interest, such that fluid permeation to make the material damp, or, conversely, evaporation to make it dry, then Lattice-Gas based models occur to be stiff, especially for parallel implementation [7]. Hence, more adaptable models are needed, which are capable to simulate all kinds of particles moving and interacting with solid pore walls.

The described version of a CA model simulates three forms of particle movements: forced convection, diffusion (surface leveling) and phase transition (evaporation). In fact, it is a discrete version of the convection-diffusion partial differential equation [8], the latter being capable to work in linear channels only. Being very simple, the proposed model is flexible enough: modification of CA transition functions allows to add the hydrophilous soaking, drainage and other processes.

The paper consists of Introduction, four sections, Conclusion, and list of references. In the Introduction the motivation of the investigation is explained. The second section presents formal definitions. In the third — the computation algorithm is given. The results of simulation are shown in the fourth (sequential version) and fifth (parallel version) sections and discussed in the Conclusion.

2 Formal Representation of a CA Model

CA is a set of identical computing units, that are represented as pairs $(u, \mathbf{x})$, called *cells*, where $u \in A$ is a *cell state* from the alphabet A, $\mathbf{x} \in X$ is a *name*, usually given by a vector $\mathbf{x} = (i, j, k)$ from a set of coordinates of a finite 3–dimensional discrete space X. A set of names

$$T(\mathbf{x}) = \{\mathbf{x}, \mathbf{x} + \mathbf{a}_1, \ldots, \mathbf{x} + \mathbf{a}_{n-1}\}, \tag{1}$$

where $\mathbf{a}_j$ is a shift vector, $n = |T(\mathbf{x})|$, is called a *template.* The cells named from $T(\mathbf{x})$ form a *local configuration*

$$S(\mathbf{x}) = \{(u_0, \mathbf{x}), (u_1, \mathbf{x} + \mathbf{a}_1), \ldots, (u_{n-1}, \mathbf{x} + \mathbf{a}_{n-1})\}. \tag{2}$$

The set of cells $\Omega = \{(u_i, \mathbf{x}_i) | u \in A, \mathbf{x} \in X, \mathbf{x}_i \neq \mathbf{x}_j\}$ is referred to as a *cellular array*, and a list of states of cells from Ω — as a CA *global state* $\Omega_A = (u_1, u_2, \ldots, u_{|X|})$.

CA functioning is determined by *operator* $\Theta(X)$ that may be a composition of several more simple operators:

$$\Theta(X) = \Phi(\Theta_1(X), \ldots, \Theta_n(X)), \tag{3}$$

which in their turn, are composed of *substitutions*

$$\theta(\mathbf{x}) : S(\mathbf{x}) \to S'(\mathbf{x}), \tag{4}$$

where $|S(\mathbf{x})| \geq |S'(\mathbf{x})|$. The first $m' = |S'(\mathbf{x})|$ cells in $S(\mathbf{x})$ comprise the *base* of $\theta(\mathbf{x})$, and the remaining $(m - m')$ cells play the role of a *context.*

A substitution $\theta(\mathbf{x})$ is applicable to a cell $(u, \mathbf{x}) \in \Omega$, if $S(\mathbf{x}) \subseteq \Omega$, a cell with a variable state being applicable, if the range of u is in A. Application of $\theta(\mathbf{x})$ implies replacing base cells states $(u_j, \mathbf{x} + \mathbf{a}_j) \in S'(\mathbf{x})$ by

$$u'_j = f_j(u_1, \dots, u_n), \quad n = |S(\mathbf{x})|, \quad j = 0, \dots, |S'(\mathbf{x}|, \tag{5}$$

where $f_j(u_1, \dots, u_n)$ – is a *transition function.* The context cells states remain unchanged.

Application $\theta(\mathbf{x})$ to all $\mathbf{x} \in X$ comprises a *global operator* $\Theta(X)$, which executes the transformation of $\Omega(t)$ into $\Omega(t+1)$, performing an *iteration.* In order to avoid conflicting situations leading up to data loss, the global operator should satisfy the following correctness condition [11]:

$$T_k(\mathbf{x}) \cap T_m(\mathbf{y}) = \emptyset, \quad \forall \mathbf{x}, \mathbf{y} \in X, \quad \forall k, m \in \{1, \dots, l\}, \quad l = |\Theta(\mathbf{x})|. \tag{6}$$

There are several modes of executing the global operator $\Theta(X)$, the main of them are synchronous and asynchronous ones.

Synchronous mode implies the following sequence of actions: (1)for all $(u, \mathbf{x}) \in \Omega(t)$ new states are computed by (5); (2) in all cells $(u, \mathbf{x}) \in \Omega(t)$ the states $u(\mathbf{x})$ are replaced by $u'(x)$. To satisfy correctness condition (6) in synchronous mode the substitutions (4) of $\Theta(X)$ should have a single cell base, i.e.

$$|S'(\mathbf{x})| = 1 \quad \forall \theta_i \in \Theta(\mathbf{x}), \tag{7}$$

Asynchronous mode implies the following way of local operator application. (1) with probability $p = 1/|X|$ a cell $(u, \mathbf{x}) \in \Omega$ is chosen; (2) $\Theta(\mathbf{x})$ is applied to a chosen cell and the base cells states $(u, \mathbf{x}) \in S(\mathbf{x})$ are immediately replaced by the corresponding $(u', \mathbf{x}) \in S'(\mathbf{x})$. By condition, it is agreed that $|X|$ repetitions of (1) and (2) comprise an iteration. Such an agreement is helpful, because it is in accordance with the synchronous mode and with a definition of a step in kinetic Monte-Carlo method [9].

Since in asynchronous CA local operator is applied sequentially, condition (6) is always met. The problem of correct computation arises only if the algorithm of asynchronous CA is executed in parallel on several processors [10,12].

Ordered asynchronous mode is also frequently used. It is a modification of the asynchronous mode, when $\theta(\mathbf{x})$ is applied sequentially to the ordered cell set.

Besides these modes any other one is also admissible, if it fits the phenomenon under simulation and satisfies (7). The mode of functioning is an essential parameter of a CA, i.e. if two CA differ only by functioning modes, then their evolutions may be quite different. Thus, the unambiguous definition of a CA is a four tuple

$$\aleph = \langle A, X, \Theta, \rho \rangle, \tag{8}$$

where ρ is the mode of operation. These four notions define the algorithm of CA functioning. But, they do not define the character of its behavior: one and the same CA with different initial global states may generate quite different evolutions. The bright example is the well known CA referred to as "Conway's Game of Life "[13]. So, a CA model of a specific process is further given as a five-tuple

$$\mathbf{A} = \langle A, X, \Theta(X), \rho, \Omega(0) \rangle. \tag{9}$$

3 The Algorithm of Simulation Water Permeation through Soil

In the CA model under consideration the superposition of several CA is used [14], which is represented as a composed CA (8), whose notions have the following interpretation.

State alphabet $A = \{0, 1, 2, 3\}$, where 0 is interpreted as a void space, 1 — as solid substance of pore walls, 2 — as a water particle, 3 — as soaked solid, inherent to hydrophilous materials.

Discrete space X is a Cartesian lattice

$$X = \{(i, j, k) : i = 0, \ldots, I; j = 0, \ldots, J; k = 0, \ldots, K\}. \tag{10}$$

Operator $\Theta(X)$ is a global composition

$$\Theta(X) = \Theta_C(X) \circ \Theta_D(X) \circ \Theta_H(X), \tag{11}$$

where $\Theta_C(X)$, $\Theta_D(X)$, and $\Theta_H(X)$ are operators of convection, diffusion, and hydrophilicity, respectively. Global superposition implies application of each global operator to the result of the preceding one.

Convection operator $\Theta_C(X)$ is in its turn a global superposition

$$\Theta_C(X) = \Theta_G(X) \circ \Theta_E(X), \tag{12}$$

where $\Theta_G(X)$ and $\Theta_E(X)$ are single substitution operators, each consisting of $\theta_G(i, j, k)$ and $\theta_E(i, j, k)$, for gravitation and evaporation, respectively.

$$\theta_G(i, j, k) : \{(1, (i, j, k))(2, (i, j, k-1))\} \xrightarrow{p_G} \{(2, (i, j, k)(1, (i, j, k-1))\} \tag{13}$$

simulates the propagation of water particles along the gravitation direction k down, letting void particles pass up. Such a movement is simulated by K sequential steps in ordered asynchronous mode, at each l-th step ($l = 1, \ldots, K$) (13) being applied synchronously to all cells having as a third coordinate $k = K - l$. Probability in (13) may be taken as $p_G = 1$, since propagation rate under gravitation is assumed to be larger than that under evaporation.

The substitution

$$\theta_E(i,j,k) : \{(2,(i,j,k))(1,(i,j,k-1))\} \xrightarrow{p_E} \{(1,(i,j,k)(1,(i,j,k-1))\} \quad (14)$$

simulates evaporation, which is a phase transition process, leading to a decrease of total water amount. Probability p_E in general case is the function of k, allowing for the differentiation of evaporation intensity in the vicinity of the surface and in deep layers in the soil. $\theta_E(i,j,k)$ is applied in ordered asynchronous mode along the kth axis, $(k = 0,\dots,K-1)$, at each k step applying $\theta_E(i,j,k)$ synchronously to all cells having $(i,j) \in \{I \times J\}$.

Diffusion operator $\Theta_D(X)$ simulates the process of density equalization, which implies leveling the free surface of water, whatever it might be: in caverns, horizontal pores and over the soil surface. In order to conform the rates of diffusion and convection operators, the first should be applied n times in sequence, n being large enough to provide water surface smoothing.

$$\Theta_D(X) = (\Theta_d(X))^n, \quad (15)$$

where $\Theta_d(X)$ contains one probabilistic substitution $\theta_D(i,j,k)$ called *naive diffusion* [15], based on a five-point local configuration

$$\begin{aligned} S(i,j,k) = \{&(u_0,(i,j,k)),(u_1,(i-1,j,k)),(u_2,(i,j+1,k)),(u_3,(i+1,j,k)),\\ &(u_4,(i,j-1,k))\} = \{(u_l(i,j,k))_l : l = 0,\dots,4\}. \end{aligned} \quad (16)$$

$\theta_D(i,j,k)$ executes the exchange of states between the cell $(2,(i,j,k))$ and one out of those its neighbors in k-th plane, whose state $u_l = 1$, i.e.

$$\begin{aligned} \theta_D(i,j,k) : \{&(2,(i,j,k))(1,(i,j,k)_l)\} \xrightarrow{p_D} \{(1,(i,j,k))(2,(i,j,k)_l\},\\ &u_l,u_0 \in \{1,2\}, \quad l = 1,2,3,4. \end{aligned} \quad (17)$$

$\theta_D(i,j,k)$ is applied sequentially to all kth planes, $k = 0,\dots,K$. In each k-th plane it is applied like a 2D asynchronous naive diffusion operator with probability $p_D = 1/m$, where m is the number of cells in $S(i,j,k) \setminus (u_0,(i,j,k))$ with void states. When pore walls are not smooth, and the neighborhood of $(u_0,(i,j,k))$ contains a pore wall cell, then $p_D < 1/m$, which implies, that some water particles may be adhered to the wall.

Hydrophilicity operator $\Theta_H(X)$ is used if the soil has hydrophilous inclusions. It simulates the process of soaking some pore wall cells during the permeation and the reverse process of drainage from soaked cell during the process of water evaporation. So, it is a superposition of two global operators:

$$\Theta_H(X) = \Theta_S(X) \circ \Theta_Q(X),$$

where $\Theta_S(X)$ simulates soaking process, and $\Theta_Q(X)$ — the drainage.

The soaking operator contains a single substitution

$$\begin{aligned} \theta_S(i,j,k) : \{&(1,(i,j,k))(2,(i,j,k)_l)\} \xrightarrow{p_S}\\ &\{(1,(i,j,k))(3,(i,j,k)_l\}\\ &(i,j,k)_l \in T(i,j,k), \end{aligned} \quad (18)$$

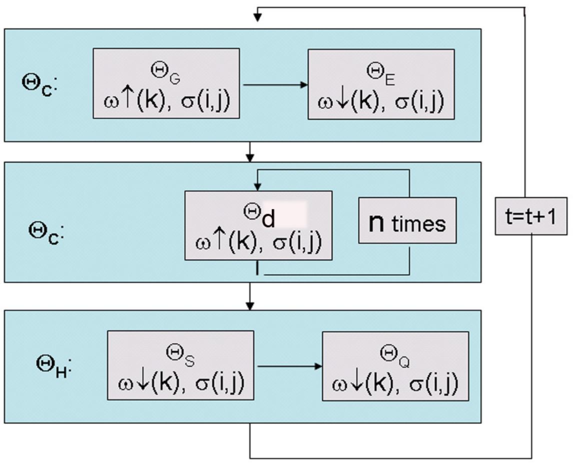


Fig. 1. The schematic representation of the algorithm of porous material damping process simulation: $\omega \uparrow (k)$ stands for ordered asynchronous mode along k-axis up, $\omega \downarrow (k)$ stands for ordered asynchronous mode along k-axis down, $\alpha(i,j)$ stands for asynchronous mode over (i,j) plane, $\sigma(i,j)$ stands for synchronous mode over (i,j) plane.

which is applied in synchronous ordered mode along kth axis and synchronously in each k-th 2D plane. The drainage operator consists of a substitution

$$\begin{array}{l}\theta_Q(i,j,k) : \{(3,(i,j,k))(2,(i,j,k)_l)\} \xrightarrow{p_Q} \{(1,(i,j,k))(2,(i,j,k)_l\} \\ \qquad\qquad\quad (i,j,k)_l \in T(i,j,k),\end{array} \tag{19}$$

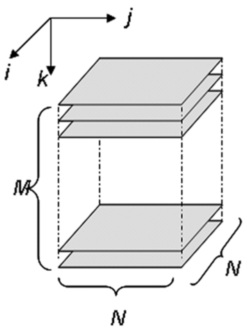


Fig. 2. A 3D cellular array structure of the porous sample

which converts a soaked cell again into solid with the probability p_Q. The probabilities p_S and p_Q depend both on soil properties and on CA parameters, and should be determined by simulation for each type of soil. The above four notions (A, X, Θ, ρ) define the composed CA, whose functioning is represented by the algorithm shown in (Fig.1). To construct the CA model of a particular task, the initial global cellular array should be known as well, which may be obtained from the data that describe the medium morphology under simulation.

4 Results of Sequential Version Implementation

In our case the initial cellular array $\Omega(0)$ is obtained by transforming the digitized representation of a sample of soil of the size $10,304 \times 10,304 \times 21,88892\,\mathrm{mm}^3$. It was packed into 1480 files, each having the size 700×700 bytes (Fig.2). So, the linear dimension of a cell is $h = 14.72$ mkm, and the cardinality of the cellular array $|X| = 725.2 \cdot 10^6$. The transformation of given files into $\Omega(0)$ was done by creation of a 3D Boolean array of size $I \times J \times K$, $(I = J = 700, K = 1480)$, whose cells $(0, (i, j, k))$ correspond to solid, and cells $(1, (i, j, k))$ correspond to pores.

A sectional view of the soil morphology is shown in Fig.3. Obtaining digitized representation of porous medium morphology is a separate problem. Usually, there are no real sample, and the digitized representation is to be synthesized, provided its characteristics are given. A few methods of pore morphology computer representation construction are known (a good review may be found in [16]). Particularly, there is a method based on totalistic CA evolution [1].

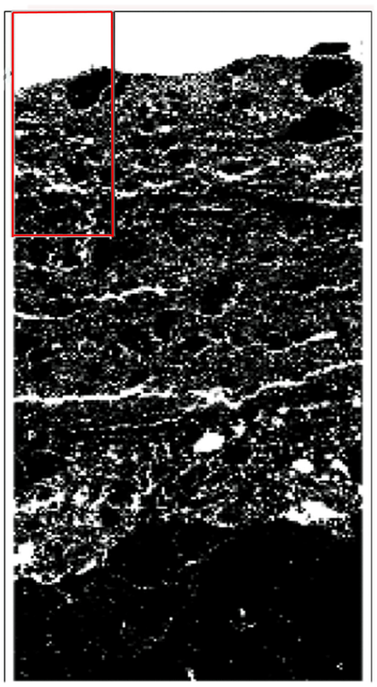


Fig. 3. The cross-section ($j = 400$) snapshot of the soil sample under investigation. The rectangle at the left top side shows the small fragment for sequential test.

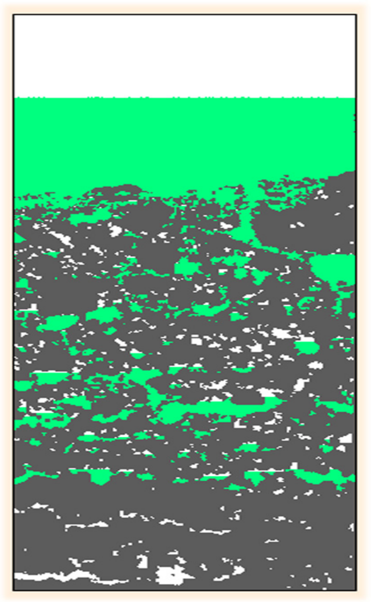


Fig. 4. The cross-section ($j = 50$) snapshot of a small fragment at $t = 50000$ with $p_E = p_H = 0$.Grey cells correspond to water.

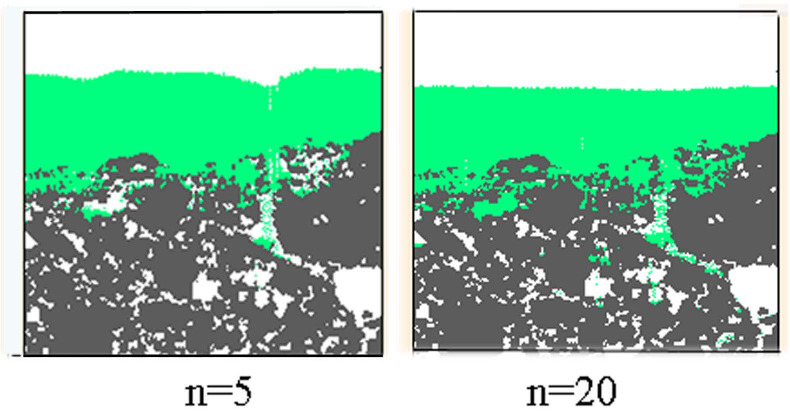


Fig. 5. The upper parts of the cross-section of the small fragment of the sample at $t = 1000$ with $n = 5$ and $n = 20$

The investigation of computational characteristics of the CA model was done in two stages. At the first stage computational experiments were performed on a small fragment of a sample using a sequential version of the algorithm (Fig.1). At the second stage capabilities of the model were studied by a series of experiments on the whole sample using 14 processors in parallel. Testing the sequential algorithm was performed on the small fragment $200 \times 200 \times 500$ (Fig.3). The program was implemented on Intel Core-i7 (2,66 GHz). Time of an iteration is 0,4 sec. The test aimed to assess water penetration rate and to range the CA model parameters values, which can be obtained only by experimental simulation. Such parameters are the following.

1. Water penetration rate is decreases in time. The depth achieved by the water during T=50000 iterations is approximately 100 cells (Fig.4. Later on permeation rate becomes very slow.

2. Coefficient n in (15), that represents the ratio of diffusion and convection intensities. Its value is obtained by running the program many times, each time increasing the value of n, until the water free surface becomes quite smooth (Fig.5). Coefficient n is the invariant of the a CA model of a concrete process [17], for each new model it should be determined over again.

In Fig.5 two snapshots of the sample fragment cross-section with $j = 50$ are shown. It is seen that with $n = 5$ the surface of the water has not enough time to become smoothed, and $n = 20$ is suitable. The value of n affects the simulation time essentially. That is why it should be taken as small as possible, but large enough to satisfy the smoothing condition.

3. The domain of possible values of evaporating probability p_E. Although the rate of drainage may be assessed basing on physical considerations, the only way of obtain p_E for a certain CA model is computer simulation. The simulation experiments (Fig.6) showed that even with $p_E = 0,01$ the process of damping is strongly affected.

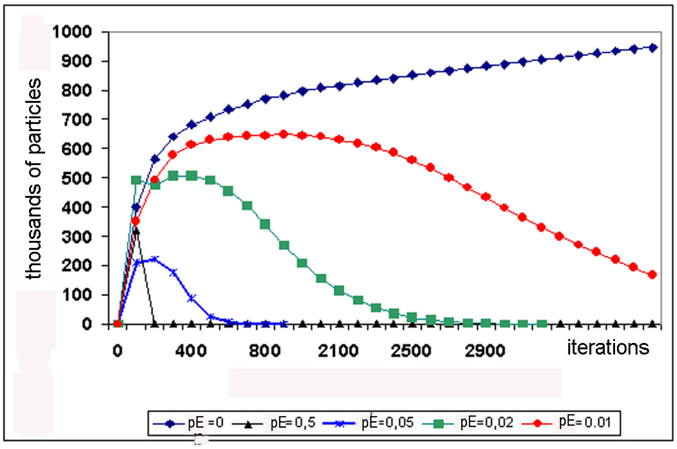


Fig. 6. Dependence of water amount in the soil with different evaporation intensity

The main peculiarity of the proposed 3D CA model is that its convection and diffusion components operate in separated dimensions: convection operates in one-dimension space along k-th axis, diffusion and soaking-drainage — in 2D planes, orthogonal to it. Actually, a method "divide and conquer" is used, that simplifies essentially the programming, especially the parallel version. It is hardly possible to assess how much this simplification decreases the accuracy of simulation, because the object of investigation cannot be precisely defined: porous materials morphology differs from sample to sample and depends from many external factors, such as dampness, temperature etc..

5 Results of Parallel Version Implementation

The architecture of the proposed model predefines the domain decomposition method for parallel processing. Convection being directed along k-th axis determines the direction of the computation domain dissection. The whole domain is decomposed into $m \times n$ subdomains. Each subdomain contains $I/m \times J/n \times K$ cells, and is allocated to a process. Interprocess data exchange is performed after each iteration in all k-th planes along i-th and j-th axes.

The well known problem of asynchronous CA parallelization necessitates to transform asynchronous CA into a block-synchronous one [10]. The procedure of such a transform is applied to all k-th planes $X_k = \{(i,j,k) : i = 0,\ldots,I; j = 0,\ldots,J;\}$, $X_k \subset X$, as follows.

1. X_k is decomposed into $|T| = 9$ nonintersecting subsets $X_k = X_k(0) \cup \ldots X_k$ in such a way, that according to correctness condition (6), intersection of their cells neighborhoods is empty.
2. At each iteration operators $\Theta_D(X)$ and $\Theta_H(X)$ are applied in nine successive stages, at each stage to all $(i,j) \in X_k(l)$, $l = 0,\ldots,8$.

3. Exchange between adjacent parts of X_k is performed after each stage.

The whole cellular array with the size $700 \times 700 \times 1480$ was decomposed into 14 domains, each of the size $350 \times 100 \times 1480$, which were allocated onto two 8-core nodes Intel Xeon 5540, 2.53 GHz (Nehalem) of the cluster NKS-30 in Siberian Supercomputer Center. Initial conditions are as follows: soil pores are empty, and over the soil there is a certain amount of water. Boundary conditions along i and j axes are periodic.

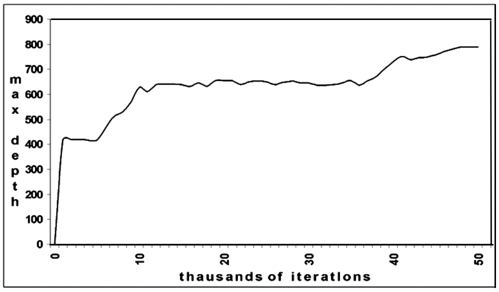


Fig. 7. Dependence of maximum permeation depth on time with $p_E = 0, p_H = 0$.

The program consists of three parts:

1. reading initial digitized sample representation from the file and dividing it into 14 subdomains;
2. executing the algorithm, computing at each iteration the required characteristics (depth of penetration, quantity of water in pores, rate of evaporation, etc.);
3. writing the obtained results into files.

The simulation experiments were performed up to T=50000 iterations aimed to investigate the evolution of permeation process under the following conditions.

- Pure permeation process without evaporation and soaking ($p_E = 0$, p_H=0), aiming to determine maximal depth water particle may reach (Fig. 7). It is seen, that the process is not monotonous due to stratified morphology of the soil.
- Permeation process affected by evaporation ($p_E = 0.001, p_H = 0$), in this case decrease of water amount in soil is also of interest, together with the maximum depth (Fig.8).
- Permeation process in the soil with hydrophilous inclusions ($p_H = 0.001$, $p_E = 0$), water amount in the soil and maximum depth of permeation (Fig.9).
- Permeation process affected both by evaporation ($p_E = 0.001$) and hydrophilicity ($p_S = 0.001, p_Q = 0$) (Fig.10).

Table 1 summarize the results.

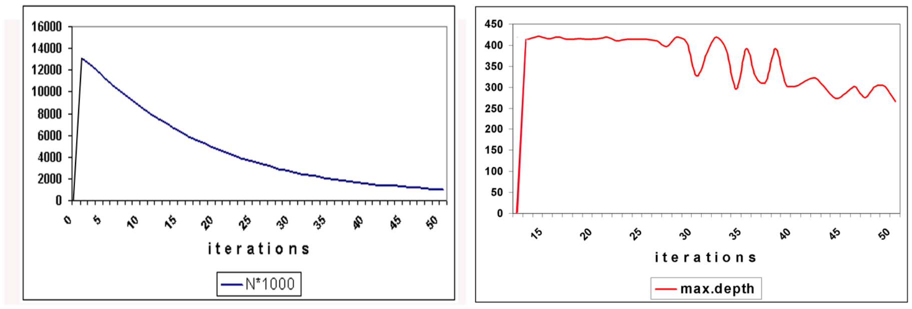


Fig. 8. Dependence of amount of water particles (thousands of particles in pores) and of maximum permeation depth on time (thousands of iterations), with evaporation intensity $p_E = 0.001$, and no hydrophilous inclusions $p_H = 0$

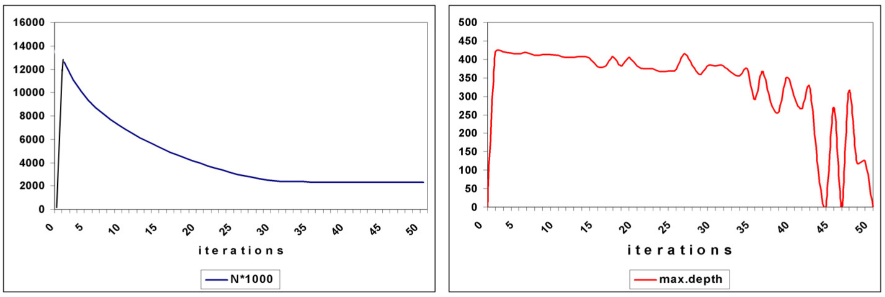


Fig. 9. Dependence of amount of water particles (thousands of particles in pores) in the soil and of maximum permeation depth) on time (thousands of iterations) without evaporation $p_E = 0$, and with hydrophilous inclusions $p_H = 0.001$.

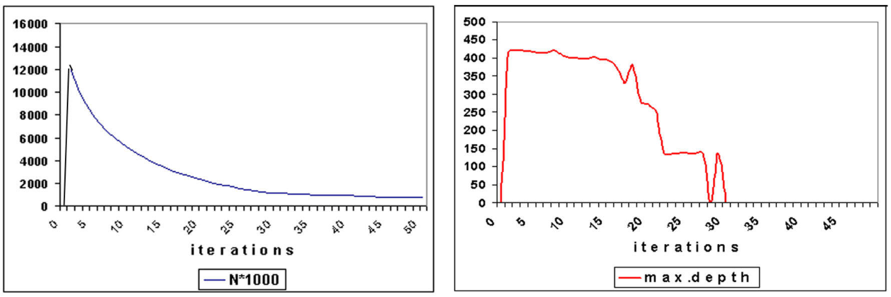


Fig. 10. Dependence of amount of water particles (thousands of iterations)in the soil and of maximum permeation depth on time with evaporation intensity $p_E = 0.001$, hydrophilicity $p_H = 0.001$.

Table 1. Maximum depth of water permeation in soil with different intensities of evaporation and hydrophilicity at T=50000

	$p_E = 0, p_H = 0$	$p_E = 0,001, p_H = 0$	$p_E = 0, p_H = 0.001$	$p_E = 0.001, p_H = 0.001$
N*1000	13572	986	2340	734
max.depth	789	267	0	0

Analysis of the above results shows that:

1. rate of water permeation decreases essentially in the deep layers of soil:
2. evaporation of water strongly affects permeation process;
3. a small amount of hydrophilous inclusions may block up the permeation.

6 Conclusion

The proposed convection-diffusion CA model of fluid permeation through porous material is remarkable by the fact that its convection and diffusion parts are separated both in time and in space. The last feature helps essentially parallel implementation, because it reduces asynchronous computations to a 2D case. The feature which makes the model advantageous relative to Lattice-Gas models is the probabilistic character of propagation operators which allows to differentiate water-solid interaction nature: sticking water particles to the pore wall, soaking and evaporation. Simulation experiments performed with the soil sample show the qualitative similarity of CA evolution to the expected behavior. To obtain quantitative characteristics in real physical values the scaling coefficients are needed. Unfortunately, they may be obtained only by studying the real material properties and comparing them to simulation results.

References

1. Bandman, O.: Using cellular automata for porous media simulation. J. of Supercomputing 57(2), 121–131 (2011)
2. Keller, L.M., et al.: 3D geometry and topology of pore pathways in Opalinus clay: Implications for mass transport. Appl. Clay Science 52, 85–95 (2011)
3. Rothman, B.H., Zaleski, S.: Lattice-Gas Cellular Automata. Simple Models of Complex Hydrodynamics. Cambridge Univ. Press (1997)
4. Hidemitsu, H.: Lattice Boltzmann Method and its Application to Flow Analysis in Porous Media. *R&D* Review of Toyota CRDL 38(1), 17–25 (2007)
5. Hoekstra, A.G., et al. (eds.): Simulating Complex Systems by Cellular Automata. Understanding complex Systems. Springer, Berlin (2010)
6. Bandman, O.L.: Cellular-Automata models of Spatial Dynamics Simulation. System Informatics (10), 58–116 (2006) (in Russian)
7. Frish, U., Hasslacher, B., Pomeau, Y.: Lattice-gas Automata for Navier-Stokes equation. Phys. Rev. Letters 56, 1505–1508 (1986)
8. Sahimi, M.: Flow phenomena in rocks: from continuum models to fractals, percolation, cellular automata, and simulation annealing. Review in Modern Physics 65(4), 1393–1534 (1993)

9. Jansen, A.P.J.: An Introduction to Monte-Carlo Simulation of Surface Reactions, `arXiv:cond-mat/0303028v1[stat-mech]` (2003)
10. Bandman, O.: 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)
11. Achasova, S., Bandman, O.: Correctness of parallel Processes. Nauka, Novosibirsk (1999) (in Russian)
12. Kalgin, K.: Parallel implementation of asynchronous cellular automama on 32-core computer. Siberian J. Num. Math. 15(1), 55–65 (2012)
13. Gardner, M.: Mathematical Games - the fantastic combinations of John Conway's new solitaire game "life". Scientific American 223, 120–123 (1970)
14. Bandman, O.: Cellular Automata Composition Techniques for Spatial Dynamics Simulation. In: Hoekstra, A.G., et al. (eds.) Simulating Complex Systems by Cellular Automata. Understanding complex Systems, pp. 81–115. Springer, Berlin (2010)
15. Toffolli, T., Margolus, N.: Cellular Automata Machines: A new Environment for Modeling. MIT Press, USA (1987)
16. Ramirez, A., Jaramillo, D.E.: Porous media generated by using an immiscible Lattice-Gas model. Computational Material Science 65, 157–164 (2012)
17. Basndman, O.: Invariants of cellular automata reactioon-diffusion models. Applied Discrete Mathematics (3), 55–64 (2012)

Temporal-Impulse Description of Complex Images Based on Cellular Automata

Stepan Belan[1] and Nikolay Belan[2]

[1] State Economic and Technologies Transport University, Kiev, Ukraine
bstepan@ukr.net
[2] “Transnistrian Radio and Television Center” OJSC
nickni@mail.ru

Abstract. This article describes method of recognition of complex images with different forms of representation. Image recognition system where image is described with impulse sequence with the help of cellular automata, used to recognize the image, has been created. The system can recognize images invariant to turning and scale variations.

Keywords: cellular automata, image, pulse sequence, recognition.

1 Introduction

Image recognition is one of the main tasks in establishment of data processing systems. Whereas all known methods operate with certain classes of images which are defined by specified set of characteristics. Characteristics are selected intuitively due to thorough analysis of images belonging to set class.

Systems built on the basis of artificial neural networks (ANN) are widely used [1-3]. Though they require a long course of learning and comprehensive preliminary data processing.

To increase accuracy and achieve real-time recognition of images cellular automata (CA) are used [4-7]. They enable efficient transformation of input image and its representation as description, which is used for its classification and recognition.

This study continues studies, results thereof are published in works [4, 7]. The study is aimed to establish method and system of recognition of images of complex visual scenes with possible organization of new classes.

2 Assignment of Characteristic Features of Images of Complex Objects on the Grounds of Multilayer Cellular Structure

The essence of recognition of images of visual scenes refers to segmentation of images into separate objects. Relations are established between images of objects, and description of visual scene takes place. It is also important to describe processed image for easy storage and search of reference image.

To achieve quick interaction and efficient description of images CAs on in-line arrangement and perform their functions are used as principle modules. Image recognition system structure is given in fig.1.

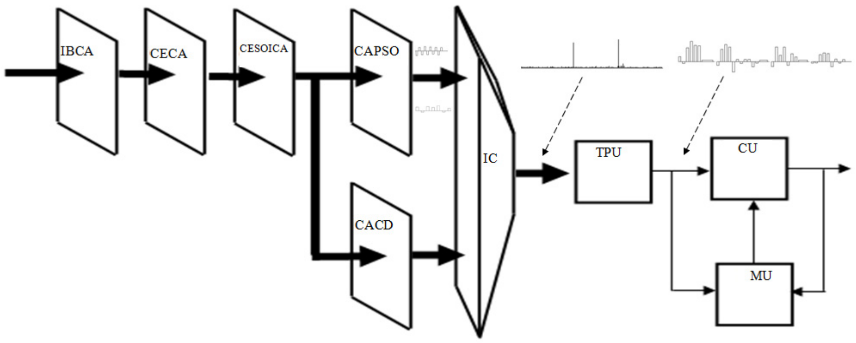


Fig. 1. Images recognition system

The system comprises cellular automata of binarization image (IBCA), cellular automata of contour extraction (CECA), consequent selection of single objects of image (CESOICA), cellular automata processing single objects (CAPSO), cellular automata calculating distances between selected objects (CACD), as well as impulse combiner (IC), thresholding unit (TPU), memory unit (MU) and comparison unit (CU).

Once image enters system input IBCA performs preliminary processing to remove noises and perform binarization. CECA selects contours of all objects of the scene in resulting image. Thereafter images are objects with the help of CESOICA (fig. 2). Fig. 2 shows examples of processes occurring in respective units with dotted lines. CACD calculates distances $L_{i,j}$ between selected objects, examples thereof is shown with dotted lines. Every object makes up a geometric structure, which is recognized by method described in studies [4,7]. These studies show structure of cellular automata and cell forming its, as well as consider process of establishing impulse consequence for one object.

The method of establishing impulse sequence for one object is as follows [4,7]. Excitation signal is distributed on CA field and, having reached the closest end cell of

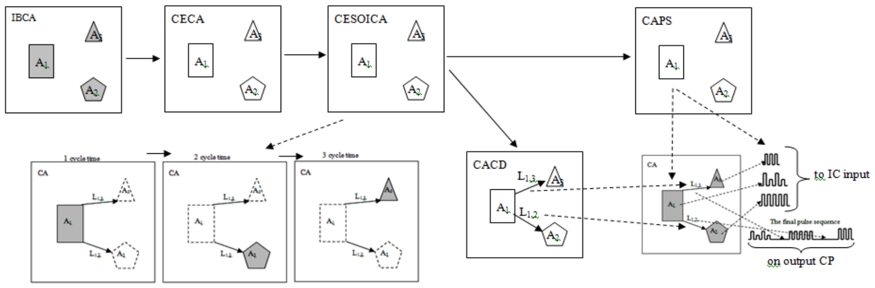


Fig. 2. Image processing in every unit

the object, starts transferring to the next closest end cell etc. While passing the contour of the object impulse is formed at CA output with every change of direction of excitation signal transfer. Impulses establish sequence, and every impulse has its own amplitude corresponding to change of direction. The bigger is the change of direction, the bigger is impulse amplitude. Thresholding removes impulses, whose amplitudes are less than set threshold, and amplitudes of remaining impulses are increased for a value corresponding to the number of neighboring removed impulses. The shape of image object is determined with regard to the number of remaining impulses, temporary durations between them and their amplitudes. Any generated sequences make up one common sequence, which describes the scene as a whole with regard to changes of scale and orientation.

In fact, at output of cellular automata forming impulses (CAPSO and CACD) impulse sequences describing every object individually are generated (fig. 2). IC ensures establishment of one impulse sequence that describes visual scene.

Result of computer modeling of method for simple shapes is given in fig. 3. Impulse sequence is shown for every separate object. Impulses with single and double amplitudes are removed. Concave tops are represented by negative impulses.

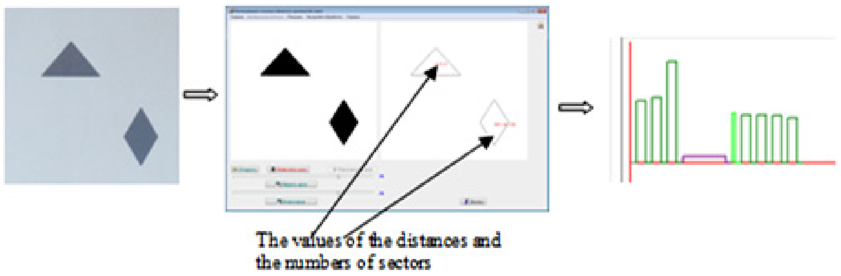


Fig. 3. Example of formation of pulse sequences for all the objects in the scene

Fragment preview total pulse sequence shown in fig. 4.

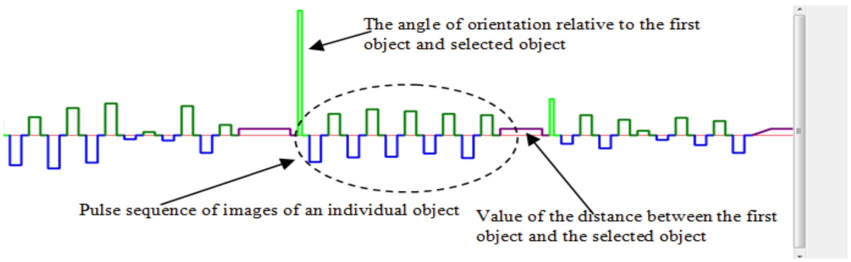


Fig. 4. Fragment preview of the pulse sequence

Relation between objects is defined with regard to time intervals between impulse sequences. If point of reference is selected strictly, additional procedures that define orientation and location are used for retuned image. That is why it is most effective to select a point of reference at the first object of impulse generation, what enables defining angle of return with regard to shift.

Thus, the whole visual scene can be presented by a set of objects, which is structurized in such a way to meet the condition

$$A_{i-1} \geq A_i \geq A_{i+1}$$

where A_i – *i*-th object in visual scene, which comprises a set of related cells $A_i = \{a_j\}$.

The object which is formed by the biggest number of cells is chosen the first. The second is the object containing less number of cells than the first one, but bigger, than the rest etc. Whereas closest distance between neighboring objects $L_{i,i+1}$ is defined. If several selected objects with equal number of cells are available, sequence of objects is defined by distance to the previous object. If $L_{i-1,i} < L_{i-1,i^1}$, the next object is A_i and $A_i{}'$ becomes A_{i+1}. Rarely, when $L_{i-1,i} = L_{i-1,i^1}$, sequence of objects is defined by location of the object clockwise.

Impulse sequence passes thresholding in TPU and comes to inputs CU and MU. CU ensures its comparison with sequence stored in MU. Impulse sequences can be represented with codes.

3 Principle of Thresholding Pulse Sequence in the System

At IC output impulse sequence is formed, which can be presented with triple

$$I = \langle Y_i, L_i, \alpha_i \rangle, \tag{1}$$

where Y_i – sequence of impulses describing *i*-th object; L_i – minimum distance between two neighboring objects defined by the number of cells; α_i – angle of orientation of *i*–th object with regard to starting point.

At first, class to which object of the scene is belonging is defined. This class is defined by basic model. It is followed by object identification by means of detailed processing of all the elements of model triple (1). At that thresholding is used for every object and limits of thresholding are set. It enables removing extra elements in the scene and selecting an object from resulting set of benchmarks.

Objects which are referred to noises can be located in any area of visual scene. Application of thresholding enables removing those objects, whose number of cells is less than set threshold. If distance between selected objects does not exceed set threshold, such objects are united in one. Space between them referred to background is filled up. Thresholding of cellular objects is accompanied with thresholding of every separate object by algorithms described in studies [4,7].

For unrecognized object it is assigned with identity code. Identity code shows establishment of the new class. This class is the first benchmark in established class. In future, while new objects related to established class arrive, the class is strengthened. If image has been recognized, this class is strengthened and threshold limits are expanded.

Such approach enables expanding classes in unlimited number. Whereas minimum basis for class establishment and identification of objects inside of the class is made up.

Images with random noise levels, black-and-white and colored, have been used as complex images (fig. 5.). The system does not process inner contours of images, as well as objects containing less than 4 pixels, though it defines their availability. Images sized 500×500 pixels of bmp or jpg format have been used. These limits will be eliminated in further studies.

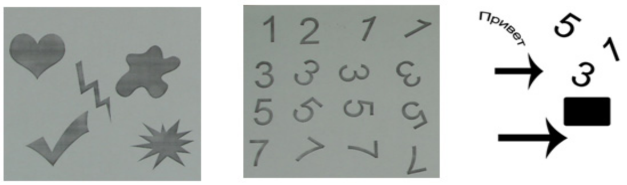


Fig. 5. Examples of images that involved in the experiment

4 Conclusion

Results of the study have shown prospectivity of CA for efficient description of images. Organization of parallel processes in the course of processing of these objects results in high-speed response. Experimental studies by means of computer modeling of image recognition process provide 93% positive results at various angles of turn and changes of scale.

References

1. Leon, O.C., Tamas, R.: Cellular Neural Networks & Visual Computing, 396 p. Cambridge University Press (2002)
2. Mohamad, H.H.: Fundamentals of Artificial Neural Networks, 544 p. MIT Press, Cambridge (1995)
3. Xingui, H., Shaohua, X.: Process Neural Networks: Theory and Applications (Advanced Topics in Science and Technology in China), 240 p. (2010)
4. Belan, S.: Specialized cellular structures for image contour analysis. Cybernetics and Systems Analysis 47(5), 695–704 (2011)
5. Bandini, S., Bonomi, A., Vizzari, G.: An Analysis of Different Types and Effects of Asynchronicity in Cellular Automata update Schemes. Natural Computing 11(2), 277–287 (2012)
6. Ioannidis, K., Andreadis, I., Sirakoulis, G.C.: An Edge Preserving Image Resizing Method Based on Cellular Automata. In: Sirakoulis, G.C., Bandini, S. (eds.) ACRI 2012. LNCS, vol. 7495, pp. 375–384. Springer, Heidelberg (2012)
7. Belan, S., Belan, N.: Use of Cellular Automata to Create an Artificial System of Image Classification and Recognition. In: Sirakoulis, G.C., Bandini, S. (eds.) ACRI 2012. LNCS, vol. 7495, pp. 483–493. Springer, Heidelberg (2012)

3D Heart Modeling with Cellular Automata, Mass-Spring System and CUDA

Ricardo Silva Campos[1], Ronan Mendonça Amorim[2], Bernardo Lino de Oliveira[3], Bernardo Martins Rocha[1], Joakim Sundnes[3], Luis Paulo da Silva Barra[1], Marcelo Lobosco[1], and Rodrigo Weber dos Santos[1]

[1] Universidade Federal de Juiz de Fora, Programa de Pós-Graduação em Modelagem Computacional, Juiz de Fora MG, Brazil
`rodrigo.weber@ufjf.edu.br`
`http://www.fisiocomp.ufjf.br`
[2] University of Calgary, Calgary, Canada
[3] Simula Research Laboratory, Norway

Abstract. The mechanical behavior of the heart is guided by the propagation of an electrical wave, called action potential. Many diseases have multiple effects on both electrical and mechanical cardiac physiology. To support a better understanding of the multi-scale and multi-physics processes involved in physiological and pathological cardiac conditions, a lot of work has been done in developing computational tools to simulate the electro-mechanical behavior of the heart. In this work, we implemented an aplication to mimic the heart tissue behavior, based on cellular automaton, mass-spring system and parallel computing with CUDA. Our application performed 3D simulations in a very short time. In order to assess the simulation results, we compared them with another synthetic model based on well-known partial differential equations(PDE). Preliminary results suggest our application was able to reproduce the PDE results with much less computational effort.

1 Introduction

Cardiac disorders are the major cause of death worldwide. Therefore the scientific community has done a lot studies in order to find the causes of heart diseases. The heart's role is pumping blood to all organs, allowing the body to change CO_2 by O_2 with lungs. The mechanical contraction of the cardiac tissue that ejects blood is preceded and triggered by a fast electrical wave, i.e. the propagation of the so called action potential (AP). Abnormal changes in the electrical properties of cardiac cells as well as in the structure of the heart tissue can lead to life-threatening arrhythmias and fibrillation.

A widely-used technique to observe the heart behavior is in silico experiments. They comprise in mathematical models that can reproduce the heart's tissue function through computational tools. Generally these models are described by differential equations, representing the cell's electrical and mechanical activity by ordinary differential equations (ODEs), and the electrical wave propagation on the tissue and cardiac mechanics via partial differential equations (PDEs). Cardiac cells are connected to each other by gap junctions creating a channel between neighboring cells and allowing the flux of electrical current in the form of ions. An electrically stimulated cell transmits an electrical signal to the neighboring cells allowing the propagation of an electrical wave to the whole heart which triggers contraction.

Although PDEs are able to perform realistic tissue simulations, they involve the simulation of thousands of cells, which make its numerical solution quite challenging. This is an issue for clinical softwares, that may demand accurate results and real-time simulations. In this manner, some effort has been done in speeding up the solving process of PDEs by parallel computing, as well different techniques to emulate PDEs with less computational cost.

This work implements a cellular automata model to represent electrical excitation of cells propagating according to simple rules in a regular, discrete and finite network. It uses precomputed profiles of cell AP and force-development that mimics those obtained by complex models based on ODEs. Although it is less accurate than the models based on ODEs, it is much faster than PDE based-simulators, making possible real time simulations of heart behavior. We present a 3D cellular automata (CA) simulator of the electrical activity of the heart coupled with a mass-spring system to simulate the cardiac mechanical behavior. Our mechanical model is governed by Hooke's Law, plus a damping and a volume preserving equation.

With our simulator we were able to evaluate interesting cases such as the influence of ischemic cells on the generation of spiral waves and the mechanical behavior under this pathological condition. In order to simulate thousands of cells we have parellelized our application with CUDA, so that the simulation runs on Graphic Processing Units. Because of the embarrassingly parallel nature of the CA and the mass-spring system, the simulator was able to allow real-time simulation for relatively large setups, i.e. for cardiac tissues composed by large number of cells.

Some electro-mechanical heart models based on cellular automaton can be found in the literature, in 2D and 3D [7,8]. The main contribution of this work is adding a preserving volume restriction to the model. This is important to get accurate simulations, since the cardiac tissue is mostly composed of water, therefore it does not have big changes in its volume. Despite its incompressibility, the tissue contraction can cause a big change in its thickness. This feature can not be simulated by simple mass-spring systems with damping[9]. Futhermore, this work is an evolution of our previous cellular automaton-based model of the heart[6], that was able only to perform 2D simulations of compressible model.

2 Methodology

The implementation of this work is related with a cellular automaton to represent the action potential propagation through the tissue, another cellular automaton to model the force development coupled with the electrical activity and the Newton's Law to model a mass-spring system. Futhermore we have added a damping coefficient and volume preserving equation. And finally, in order to allow simulations with a lot of elements, we proposed a parallel implementation with CUDA.

2.1 Cellular Automata

A cellular automaton is the model of a spatially distributed process that can be used to simulate various real-world processes. A 2-dimensional cellular automaton consist of a two-dimensional lattice of cells where all cells are updated synchronously for each discrete time step. The cells are updated according to some updating rules that will depend on the local neighborhood of a particular cell.

The Cellular Automata (CA) can be used to simulate macroscopically the excitation-propagation in the cardiac tissue. If the electrical potential of a cell exceeds a threshold the cell gets excited and this can trigger the excitation of the neighboring cells. Therefore, an electrical stimulus will propagate through the CA grid as the time steps are computed.

The following sections will describe the CA approach for simulating the anisotropic electrical propagation in the cardiac tissue and the force-development in each cell. In this work, the CA states are related to the action potential (AP) and force development in a cell. To make CAs more efficient they are usually parametrized using simulated data from accurate models. This means that the states related to the AP in the cell will be related to a specific portion of the cardiac cell potential. Figure 1 presents the AP computed by ODEs, the AP divided into five different states and an interpolation of the discrete values of AP computed by CA. The interpolation granted smooth propagations of the action potential with low computational costs.

In state S0 the cell is in its resting potential where it can be stimulated, in S1 the cell was stimulated and can stimulate the neighbors. In S2 the cell is still able to stimulate the neighbors. In S3 the cell is in its absolute-refractory state where it cannot be stimulated and does not stimulate its neighbors. In S4 the cell is in its relative refractory state where it can be stimulated by more than one neighbor but it does not stimulate its neighbors. As described, the states of a cell generate rules for when a cell can stimulate a neighbor and when it can be stimulated. These rules are an important aspect which will allow the stimulus to propagate. Another important point is how the cells change their states. The AP has a predetermined period so that the states will be spontaneously changed when an AP starts, according to the timing presented in Figure 1.

Some fatty build-ups in arteries can totally or partially block the blood irrigation in parts of cardiac tissue causing it to die (not propagating stimulus, dead cells) or to behave differently (ischemic cells). The ischemic cells of the tissue present an AP with different properties than a healthy cell. Usually it has a shorter duration of the AP with different potential values. Table 1 presents the CA states, potential and times for both healthy and ischemic tissue. The contraction of a cardiac cell is coupled with the electrical potential of the cell.

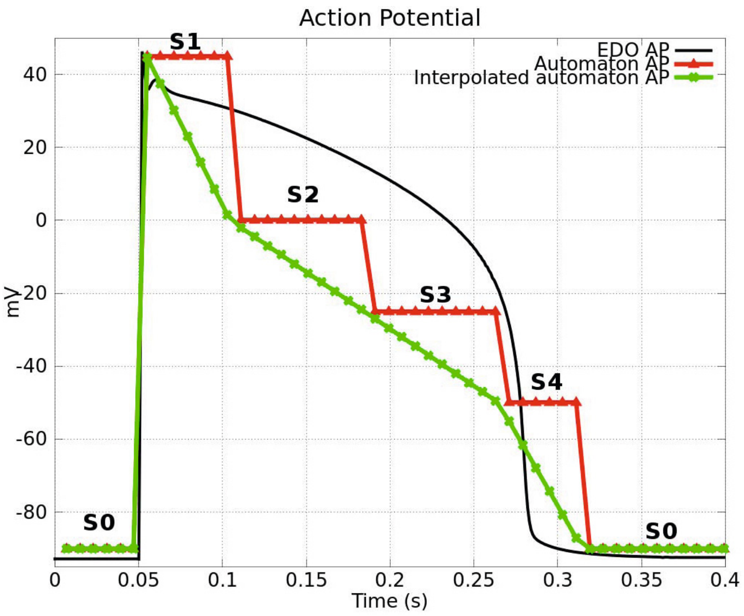


Fig. 1. Action potential of a cardiac cell:The black line is the AP computed by ordinary differential equations. The red line is the AP separated into five different states (S0, S1, S2, S3, S4) for the cellular automata, and the green line is the interpolation of the discrete values of the CA-computed AP.

When the cell is stimulated, there is an increase in the concentration of calcium ions inside the cell, which trigger the cell contraction. The force development has a delay after the cell is stimulated because of its dependence on the calcium ions. The force development of a cell can be represented in states that change over time like the electrical potential states. Figure 2 presents the force development states and its relation with the electrical states. The force-development states will only pass from state F0 (no contraction force) to state F1 when the electrical state of the cell goes from state S1 to S2. After this change, force development will be time dependent but will not depend on the electrical state of the cell.

Propagation Rules. The electrical propagation velocity in the cardiac tissue is dictated by the fiber direction and tissue type, and it has three axis of anisotropy. Each one has a different electro-mechanical behaviour. Figure 3 shows a ventricular tissue segment, where the dark-gray lines are the three axes. The first direction is the fiber axis, where the wave propagation velocity is faster. The sheet direction is orthogonal to the fiber, and the third direction is called sheet-normal, that is normal to the plane formed by the fiber and sheet axis.

Since the anisotropy must influence the AP propagation of the CA, we set up a velocity of propagation to each direction. So we find the direction of the neighbor element, in order to compare its direction with the fibers and then determine

Table 1. Healthy and ischemic states of the cellular automata and the respective duration and potentials

	Healthy Cell		Ischemic Cell	
CA State	Duration	Potential	Duration	Potential
S0	in rest	-90mV	in rest	-70mV
S1	50ms	+20mV	50ms	0mV
S2	80ms	0mV	40ms	-40mV
S3	80ms	-25mV	25ms	-60mV
S4	50ms	-50mV	10ms	-65mV

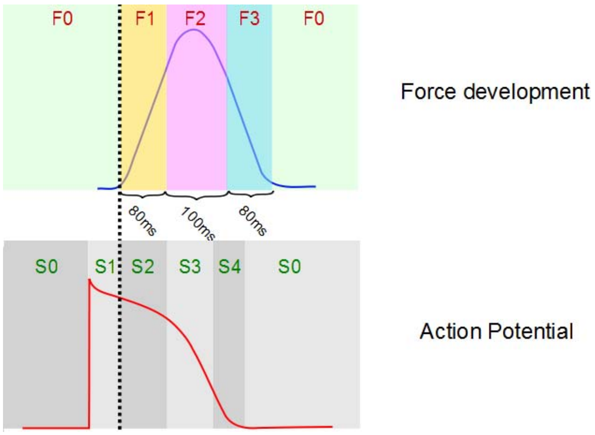


Fig. 2. Force development states in relation with the cell electrical states (Adapted from [6])

the resultant velocity. Therefore, the traveling time of a stimulus from a cell to another is found, and this depends on the directions of the propation.

The activation of a cell will depend on the time of the activation of its neighbors. For each time step each cell checks if the neighboring cells are activated. If an activated cell is found the time of the stimulus to travel from that cell is computed and compared with the activation time of the same cell to check if there was enough time to the stimulus to travel. For the activation to take place the neighboring cells must also be in state S1 or S2, i.e. states that allow one cell to stimulate another. After a cell is stimulated it will, independently of the neighboring cells, dynamically change its state until it finally goes to the states S4 and S0, when it may be stimulated again. With this set of rules the stimulus is able to propagate through the CA simulating the electrical propagation of APs on the cardiac tissue.

The CA is discretized in space and time. The 3D CA is composed by small cubes with edges $dx \times dx \times dx$, where dx is the space discretization. The vertices of the cube are the masses. CA states are updated at every discretized time, dt. Based on this information and the velocities we can calculate the time that a

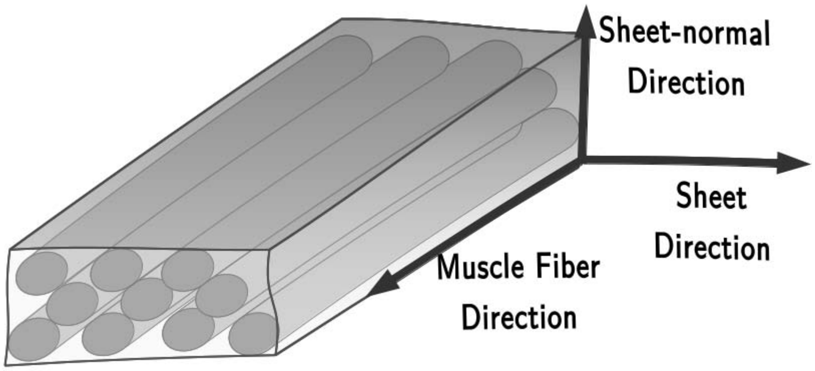


Fig. 3. Fiber directions in cardiac tissue

stimulus takes to travel from one CA cell to another. For simplification, imagine that the propagation has the same velocity v in all directions (isotropic tissue). To find the time t for a stimulus travel from the center of one cell to another, first we have to find the directions of the neighbor:

$$\overrightarrow{Dir}_{c\_n} = \frac{\overrightarrow{X}_n - \overrightarrow{X}_c}{\|\overrightarrow{X}_n - \overrightarrow{X}_c\|} \tag{1}$$

Where $\overrightarrow{Dir}_{c\_n}$ is the neighbor direction relative to the current element, $\overrightarrow{X}_n$ and $\overrightarrow{X}_c$ are the coordinates of the neighbor and current element.

We have the three vectors of the anisotropy axes, fiber, sheet and normal-sheet: $\overrightarrow{F}$, $\overrightarrow{S}$ and $\overrightarrow{NS}$, as well are given three scalars representing the velocities on each one of axis, respectivelly: vel_f, vel_s and vel_{ns}. So, to calculate the velocity of the wave propation between two elements, we use the dot product:

$$V_f = vel_f(\overrightarrow{Dir}_{c\_n} \cdot \overrightarrow{F}) \tag{2}$$
$$V_s = vel_s(\overrightarrow{Dir}_{c\_n} \cdot \overrightarrow{S}) \tag{3}$$
$$V_{ns} = vel_{ns}(\overrightarrow{Dir}_{c\_n} \cdot \overrightarrow{NS}) \tag{4}$$
$$\tag{5}$$

Once we have the vector $\overrightarrow{V} = [V_f \; V_s \; V_{ns}]$ we find the final velocity by applying a norm to $\overrightarrow{V}$:

$$v = \|\overrightarrow{V}\| \tag{6}$$

To find the distance between the elements, we also apply a norm:

$$d = \|\overrightarrow{X}_n - \overrightarrow{X}_c\| \tag{7}$$

Finally, the traveling time from one element to another is found:

$$t = \frac{d}{v} \tag{8}$$

Assuming a Moore neighborhood for 2D with radius 1, the immediate top, bottom, left, right and all diagonals are considered neighbors. For the diagonals, a different distance of $dx\sqrt{2}$ is assumed, as presented in Figure 4 A. For a 3D CA, the definition of Moore neighborhood is analogous. In Figure 4 C, we have a central element (in grey) and its 27 neighbors (in black).

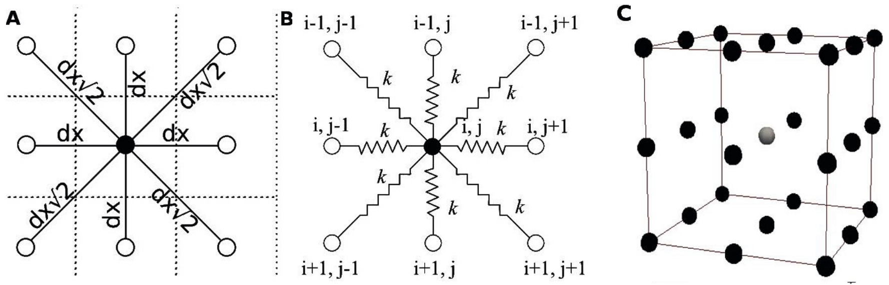


Fig. 4. A) The distances in Moore neighborhood of a cell with radius 1. B) Interconnected set of masses by springs. C) 3D Moore Neighborhood.

2.2 Mechanical Model

The modeling of cardiac tissue deformation can be simplified by the use of mass-spring systems. In such systems, masses are connected with the neighboring masses by springs and forces can be applied to the system deforming its spatial distribution. The springs of the system will try to bring the system to its initial configuration again. The cardiac tissue does not have a linear stress-strain relation. However the linear model of the Hooke's law can be used as a simplification. For each mass, the total force is the summation of the Hooke's law between each one of its connecting neighbors:

$$\overrightarrow{F}^i_s = \sum_{j=0}^{n} (-k\overrightarrow{\Delta^j}) \tag{9}$$

where k is the spring constant that determines the stiffness of a spring, n is the number of neighbors and $\overrightarrow{\Delta^j}$ the displacement between mass i and its neighbor j. But such systems will oscillate forever. In practice there will be forces in the environment that will resist to the movement. Such forces are called damping forces $\overrightarrow{F}_d$ and are proportional to the velocity of the mass $\overrightarrow{V}^i(t)$:

$$\overrightarrow{F}^i_d = -\beta\overrightarrow{V}^i(t) \tag{10}$$

where β is the damping coefficient. The interconnected set of masses (lattices) are depicted in 4 B. We also included in the system the contraction force of the cell $\overrightarrow{F}_c$, and another force in order to preserve the volume.

The preserving force is important for accurate simulations. Since the cardiac tissue is mainly composed by water, it does not have significant changes in its volume. To add this feature in the mass spring system, we applied a force to each mass in order to keep the same volume during the simulation, although the tissue contracts. This force depends on each cube surroundding the mass. It emanates from the cube's baricenter $\overrightarrow{X}_b$, and its module depends on the cube's volume variation. The resulting formula is given below:

$$\overrightarrow{F}^i_v = \sum_{\forall j \in \Omega i} k_v \frac{(V^j_t - V^j_0)}{V^j_0} \frac{\overrightarrow{X}_i - \overrightarrow{X}^j_b}{\|\overrightarrow{X}_i - \overrightarrow{X}^j_b\|} \tag{11}$$

where V^j_t is the volume of cube j sorrounding mass i with coordinates X_i. V^j_0 is the initial volume. The cubes on the neighborhood of mass i are represented by Ωi. The total force on each mass is:

$$\overrightarrow{F}^i_{total} = \overrightarrow{F}^i_s + \overrightarrow{F}^i_d + \overrightarrow{F}^i_c + \overrightarrow{F}^i_v \tag{12}$$

From Newton's second law we have that $F = ma$ and equating this with the total force we have:

$$\overrightarrow{a}^i(t) = \frac{F^i_{total}}{m} \tag{13}$$

Finally it is necessary to integrate the system of equations for each cell in the CA to simulate the mechanical deformation of the tissue. Using the Forward Euler's method we have

$$\overrightarrow{V}^i(t + dt) = \overrightarrow{V}^i(t) + \overrightarrow{a}^i(t)dt \tag{14}$$

$$\overrightarrow{P}^i(t + dt) = \overrightarrow{P}^i(t) + \overrightarrow{V}^i(t)dt \tag{15}$$

where $\overrightarrow{V}^i(t)$ and $\overrightarrow{P}^i(t)$ are the velocity and position of a mass at time t. In this way, CA and mass-spring models are coupled by F_c, i.e by the force generated by the cell during the dynamically change of its AP.

There are different approaches for modeling the electro-mechanical behavior of the heart. In [2], a method based on CA is presented that can simulate electrical propagation in 3D cardiac tissue with arbitrary local fiber orientations. In [3], a CA is used to simulate cardiac electrical propagation and a model based on the finite element method is used to simulate cardiac mechanics. In [4] the CA approach is used for the electro-mechanical simulation of the cardiac tissue. A comparison of models for cardiac tissue based on differential equations and on CA is presented in [5].

2.3 Parallel Computing

Our CA coupled with mass-spring system is not computational expensive. A mesh with a thousand elements can be simulated in a few minutes. However, our intentions for future work is to perform simulations for the whole heart, which may demand millions of elements. Futhermore, we also need to estimate some parameters using genetic algorithms (GA), in order to find accurate simulations. For this GA, it is necessary to evaluate a few hundreds of the CA, with different parameters. According to the comparison to real data, the parameters that resulted in the best outcome are chosen for the next iteration, and the other parameters are adapted. This process is repeated until the best solution reachs an acceptable error. Undoubtedly, this technique is computational expensive and may take some hours to get finished. So, in order to speedup the simulation, we proposed a parallel implementation using Graphic Processing Units (GPU) with CUDA (Compute Unified Device Architecture). In this architecture, it is necessary to copy the elements' matrix that represents the CA from the CPU (host) to the GPU (device). This is done in the beginning of the computation as follows:

```
//copy data from CPU to GPU
copyHostToDevice();
//set up the number of threads to each block
dim3 threads(3, 3, 3);
dim3 grid_size; //set up the grid
grid_size.x = (width + threads.x)/ threads.x;
grid_size.y = (height + threads.y)/ threads.y;
grid_size.z = (depth + threads.z)/ threads.z;
while(t<finalTime){
   time += dt; //increments simulations time
   simulate<<<grid_size, threads>>>();
}
```

In this code, the CA matrix is divided into blocks containing $3\times3\times3$ threads. The distinct blocks form the matrix grid that is set up according to its dimensions, *height*, *width* and *depth*. This pseudo code is run in the CPU. The functions called with the triple angle bracket are invoked by the CPU but they actually are run in the GPU.

3 Results

Different tests with the implementation were performed. The simulations used a cubic tissue with $9 \times 9 \times 9$ cells with a spatial discretization of $0.001m$ and a time discretization of $1ms$. Conduction velocity is assumed to be $0.5m/s$, $0.17m/s$ and $0.17m/s$ along the fiber, sheet and normal sheet axis, respectively. A stimulated cell is placed in the left-bottom of the tissue and the propagation will be studied. The first test assumed horizontal fiber orientation for all cells of the tissue, and all cells in the system are healthy. The tests' results are presented in Figure 5.

The second test also assumed a horizontal fiber, but it was included ischemic cells. At rest, healthy cells have $-90mV$, identified in the figures by the grey

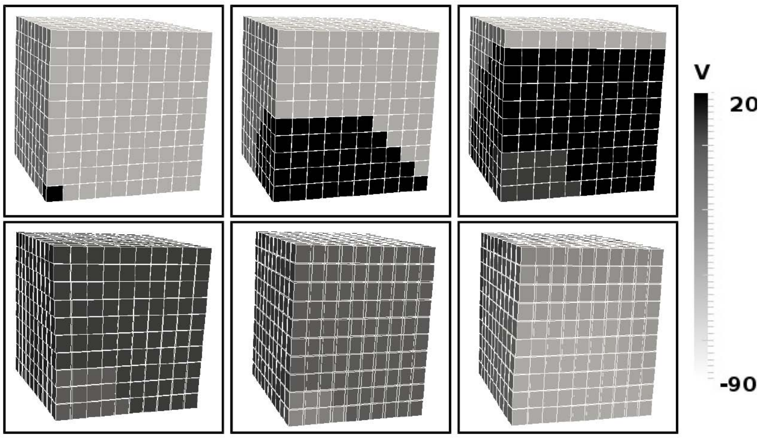


Fig. 5. Simulation of cube with $10mm$ edge, all cells are healthy and the stimulation starts in left-bottom corner, and fiber is horizontal

color. Ischemic cells have $-70mV$ and at their rest and they are colored with a darker grey shade. This configuration is presented in Figure 6. Because of its shorter AP duration the ischemic tissue is able to be stimulated again earlier than the healthy tissue. This leads to the creation of a reentry circuit where the tissue keeps stimulating itself forever via the propagation of spiral wave. This causes arrhythmic contractions of the cardiac tissue.

The third and fourth tests compared the influence of the fiber's direction in the AP propagation. In the first row in Figure 7, the AP propagates through a tissue with all tissue with in horizontal direction. In the second row, the fiber is

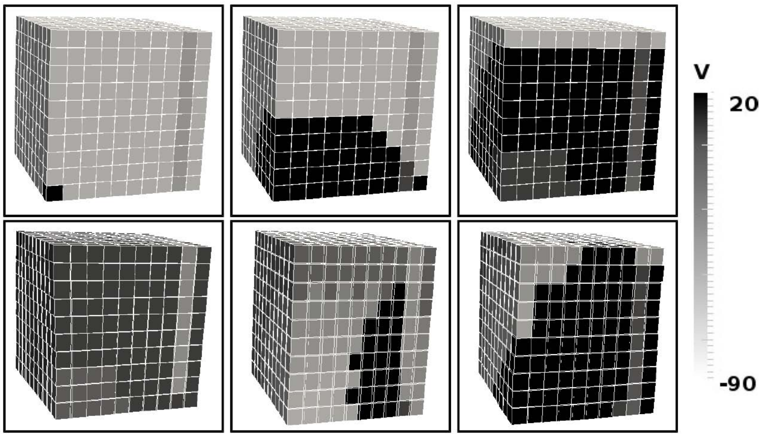


Fig. 6. Simulation with ischemic cells

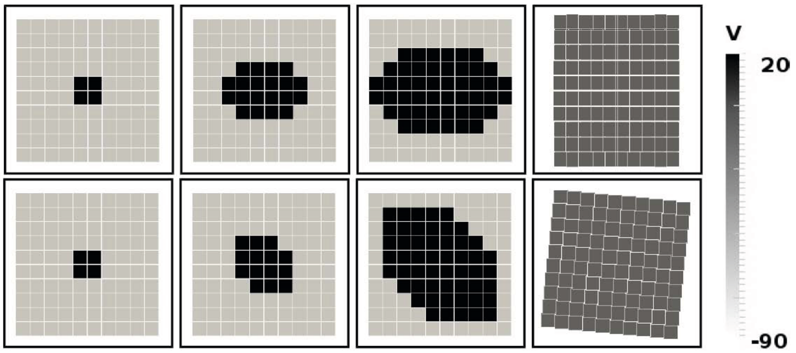


Fig. 7. Simulating with different fiber directions. First row with fibers at 0° and second row at 45°.

at 45°. It can be observed that the stimulus propagates faster through the tissue in the preferential direction of the fiber, and the mechanical contraction is also fiber-prefered. Another test was performed to evaluate the fiber influence on the AP propagation, where the same tissue was mapped with two different fibers. In Figure 8, the upper half of the tissue has horizontal fibers and the bottom half has a 45° inclined fiber. The stimuling cell is central, and the wave propagates according to fiber direction on each cell.

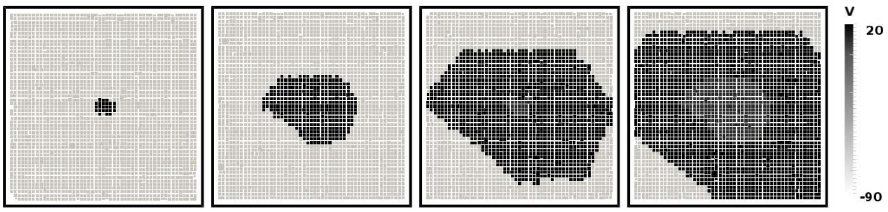


Fig. 8. Same tissue with different fibers directions

In order to compare the CA results with differential equations (DE), we have evaluated the wall thickness (WT) of the tissue during simulation and the execution time. Here, we calculate the WT as the average thickening of the tissue wall in the transmural direction. Both simulations produced a maximum of 35% WT. The implementation of DE solver is described in [10]. Briefly, it contains an eletromechanical cell model that computes electrophysiology and active force generation, coupled via calcium concentration, and the action potential propagation through the tissue is modeled with the bidomain equations.

;Comparing the execution times, we perform a test with the same configuration for both CA and DE. With a $10mm^3$ mesh, the CPU execution time for DE is 13h30min(48600s). On the other hand, the CA execution in CPU took 38s and the GPU version 7s, that respectivelly are 1200 and 6800 times faster than the simulation based on DE.

The volume preserving force (F_v) was able to decrease the volume variation. In a compressible tissue, the volume could change until 40% of its initial value. With the incompressible force, it changed at the most 10% of its initial volume. Figure 9 presents how F_v acts for volume preserving. There are two tissues configurations: compressible (part A) and incompressible (part B). F_v acts from the baricenter of each component cube, and resultant force is best visible at the mesh's corners. After a contraction in a direction, the tissue "growns" in the other directions in an attempt to keep the volume constant.

In order to our mass-spring system to reproduce a DE simulation is necessary determine parameters, which is tiresome trial-and-error process, altough it was possible to get similiar results. Nevertheless, it is still necessary to find a better technique to estimate parameters.

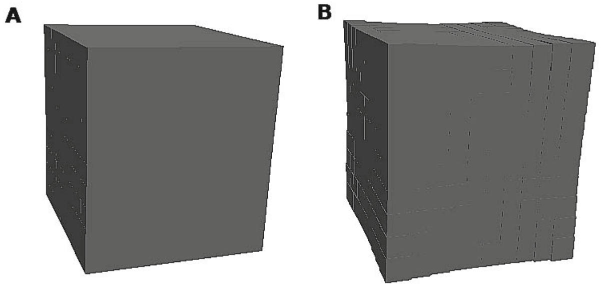


Fig. 9. Comparison between compressible (A) and incompressible (B) tissues

Finally, we studied the impact of parallelizing the application. All tests were performed in an Intel i7 machine with 8 GB of RAM, Nvidia GeForce GTX 480, running Ubuntu/Linux 4.6.1 with gcc 4.6.1. We run both implementations (sequential CPU and parallel GPU) with different meshe sizes. We simulated $1s$ of the tissue activity, all tests were run four times. We present the average execution time, in seconds and in all cases the standard deviation observed in the execution times was less than 1%. The speedup was computed considering the average execution time for each mesh size:

$$Speedup = \frac{T_{CPU}}{T_{GPU}} \tag{16}$$

Table 2 presents the performance of the sequential and CUDA codes. It is possible to observe that the speedup is bigger with bigger meshes. This happens because there is an overhead for managing threads and GPU calls. There is also an implicit barrier at end of iteration, since it is necessary to synchronize the Euler integration method where one iteration is dependent on the previous one. When small meshes are run, the overhead is not negligible comparing with their small execution time.

Table 2. Performances

Mesh size	$10mm^3$	$30mm^3$	$50mm^3$
GPU avg. time(s)	7	148	631
CPU avg. time(s)	38	1190	5697
Speedup	5.4	8.0	9.0

4 Conclusion

This work presented a 3D simulator of the electro-mechanical activity of cardiac tissue via the coupling of CA and mass-spring models. Although models based on PDEs are more accurate and detailed, they are very computationally expensive. CA has shown to be an alternative for real-time simulation because of its fast performance, that resulted in a 1200 to 6800-fold improvement on computational time. The pattern of propagation obtained with CA has shown to be very similar to the patterns obtained with PDE-based models, including WT and volume, although a more detailed comparison is necessary. The tissue deformation obtained with the mass-spring system has shown to be very responsive to the force-development providing a qualitative demonstration of cardiac contraction. The volume preserving force was able to avoid big changes in the tissue without changing its WT, although manual determing parameters for reproducing PDE simulations is a hard task. The parallel implementation speeds up the simulations by 9 times, but it is still necessary to investigate improvements for the code to decrease overheads for faster simulations.

As future work, some improvements can also be achieved in terms of determing parameters of both CA and mass-spring systems, adjusting them to data obtained in experiments or from more realistic heart models.

References

1. Campos, R.S., Amorim, R.M., Costa, C.M., de Oliveira, B.L., Barbosa, C.B., Sundnes, J., dos Santos, R.W.: Approaching cardiac modeling challenges to computer science with CellML-based web tools. Future Generation Computer Systems 26, 462–470 (2010)
2. Feldman, A.B., Murphy, S.P., Coolahan, J.E.: A method for rapid simulation of propagating wave fronts in three-dimensional cardiac muscle with spatially-varying fibre orientations. Engineering in Medicine and Biology (2002)
3. Cimrman, R., Kroc, J., Rohan, E., Rosenberg, J., Tonar, Z.: On coupling cellular automata based activation and finite element muscle model applied to heart ventricle modelling. In: 5th International Conference on Simulations in Biomedicine - Advances in Computational Bioengineering (2003)
4. Hurmusiadis, V.: Virtual Heart: Cardiac Simulation for Surgical Training & Education. In: Workshop & Conference on Virtual Reality and Virtual Environments (2007)

5. Sachse, F.B., Blümcke, L.G., Mohr, M., Glänzel, K., Häfner, J., Riedel, C., Seemann, G., Skipa, O., Werner, C.D., Dóssel, O.: Comparison of macroscopic models of excitation and force propagation in the heart. Biomed. Tech. (Berl.) 47, 217–220 (2002)
6. Amorim, R.M., Campos, R.S., Lobosco, M., Jacob, C., dos Santos, R.W.: An Electro-Mechanical Cardiac Simulator Based on Cellular Automata and Mass-Spring Models. In: Sirakoulis, G.C., Bandini, S. (eds.) ACRI 2012. LNCS, vol. 7495, pp. 434–443. Springer, Heidelberg (2012)
7. Bora, C., Serinağaoğlu, Y., Tönük, E.: Electromechanical heart tissue model using cellular automaton. In: 2010 15th National Biomedical Engineering Meeting (BIYOMUT), pp. 1–4 (2010)
8. Bora, C.: Cellular Automaton Based Electromechanical Model of the Heart (Master Thesis), Middle East Technical University (2010)
9. Mollemans, W., Schutyser, F., Van Cleynenbreugel, J., Suetens, P.: Tetrahedral mass spring model for fast soft tissue deformation. In: Ayache, N., Delingette, H. (eds.) IS4TM 2003. LNCS, vol. 2673, pp. 145–154. Springer, Heidelberg (2003)
10. de Oliveira, B.L., Rocha, B.M., Barra, L.P.S., Toledo, E.M., Sundnes, J., dos Santos, R.W.: Effects of deformation on transmural dispersion of repolarization using in silico models of human left ventricular wedge. Int. J. Numer. Meth. Biomed. Engng. (2013), doi:10.1002/cnm.2570

Towards the Introduction of Parallelism in the MakkSim Pedestrian Simulator

Luca Crociani, Giuseppe Vizzari, and Stefania Bandini

CSAI - Complex Systems & Artificial Intelligence Research Center
University of Milano-Bicocca, Viale Sarca 336, 20126, Milano, Italy
{luca.crociani,vizzari,bandini}@disco.unimib.it

Abstract. MakkSim is an agent–based pedestrian dynamics simulator extending the CA floor field model considering the influence of groups. This paper presents a systematic analysis and discussion of the different options for the introduction of parallel computation approaches in MakkSim. We briefly introduce the model, then we analyse the margins for improvement in the different phases of the simulation cycle. The alternatives for the parallelisation of the simulation engine are finally evaluated and discussed.

Keywords: Agent Systems, Crowd Simulation, Thread-Level Parallelism.

1 Introduction

Computer models for the simulation of crowds are growingly investigated in the academic context and adopted by decision makers employing commercial off-the-shelf simulators[1]. These models can be roughly classified into three categories that respectively consider pedestrians as *particles subject to forces* (see, e.g., [4]), *states of cells* in which the environment is subdivided in Cellular Automata (CA) approaches (see, e.g.,[8]), or *autonomous agents* (see, e.g. [1,5]) acting and interacting in an environment. Recent developments in this area focus on phenomenologies not previously considered by other pedestrians and crowd models: in particular, MakkSim [9] includes groups as first class entities influencing pedestrian behaviour. Members of groups strive to stay close to each other even in local high-density situations. The MakkSim model presents an adaptive component allowing groups to preserve their cohesion but the perception model of the agents is more sophisticated and it implies a higher computational cost.

CA are a parallel computing model and due to this intrinsic feature they are suitable to be effectively implemented on parallel computers achieving also a high level of performance, employing specialised hardware execution platforms (e.g. [3]), or common multi core CPUs (see, e.g., [2]). This work, building on this kind of experience in the parallelisation of simulation models, will present a systematic analysis of the different options for the introduction of parallel computation approaches in the MakkSim simulator. The paper briefly introduces the model, then it analyses its current performance and margins for improvement

[1] see `http://www.evacmod.net/?q=node/5` for a list of available simulation platforms.

in the different phases of the simulation cycle. The various alternatives for the actual parallelisation of the simulation engine are then introduced and evaluated.

2 MakkSim Overview

The main elements of the MakkSim model are the simulated *environment*, the population of *agents* representing pedestrians and their *updating strategy.*

Environment – MakkSim is discrete both in space and time. The environment is represented as a grid of squared cells with side of 40 cm, respecting the average space occupied by a pedestrian. We adopted a *floor field* approach [8] supporting agents' navigation in the environment. Three kinds of fields are defined: (i) the *path field* indicates for every cell the distance from one destination area, driving pedestrians towards it (static); (ii) the *obstacles fields* describes for every cell the distance from neighbour obstacles or walls (static); (iii) the *density field* indicates for each cell the local pedestrian density at the current time-step (dynamic).

Agents – The life-cycle of a MakkSim agent is divided into four steps: *perception, utility calculation, action choice* and *movement.* The *perception* provides to the agent all the information needed to evaluate the desirability of a cell. The choice of each action is based on an utility value assigned to every possible movement according to the following function:

$$U(c) = \frac{\kappa_g G(c) + \kappa_{ob} Ob(c) + \kappa_s S(c) + \kappa_c C(c) + \kappa_i I(c) + \kappa_d D(c) + \kappa_{ov} Ov(c)}{d}$$

The first three components employ information derived by floor fields and they consider (i) goal attraction, (ii) geometric and (iii) social repulsion. The fourth and fifth components model the social attraction element of the pedestrian behaviour, by increasing the utility of positions closer to group members. We consider two types of group: *simple* (iv), that indicates any small group characterised by a strong and simply recognisable cohesion (e.g. family or friends); *structured* (v), a larger group, that shows a slight cohesion but that is naturally split into subgroups (e.g. tourists). Finally, two special factors consider two additional phenomenologies: (vi) helps an agent to maintain the current direction, while (vii) allows to move to a cell already occupied by one pedestrian in high density conditions. An adaptive mechanism is also defined to tune the parameters of this utility function, in order to preserve the cohesion of simple groups even in high density situations. After the utility evaluation for all the cells in the neighbourhood, the choice of action is stochastic, with the probability to move in each cell c as (N is the normalisation factor): $P(c) = N \cdot e^{U(c)}$. On the basis of $P(c)$, agents chooses a cell according to their set of possible actions, defined as list of the eight possible movements in the Moore neighbourhood.

Update Strategy – The duration of each time step is 1/3 of a second, generating a linear pedestrian speed of about 1.2 ms^{-1}, consistent with the literature. Regarding the update mechanism, three different strategies are usually applied [7]: *ordered sequential*, *shuffled sequential* and *parallel* update. The first two strategies employ a sequential update of agents, respectively managed according to a *static* or *dynamic* (random) list of priorities, preventing conflicts. On the contrary, the parallel update calculates the choice of movement of all the pedestrians at the same time, managing conflicts and actuating choices in a latter stage. In previous applications [9] we adopted the shuffled sequential strategy, but we are investigating the possibility to adopt a parallel approach, since it has shown a greater adequacy in considering actual conflicts among pedestrians [6]. Moreover, this update strategy allows considering separately decisions about movement, that take place at the same time, from the actual updating of the pedestrians' positions, that must consider and solve conflicts. The first phase can be easily and conveniently parallelised, as we will show in the following.

Analysis of Simulation Cycle Computational Costs – To evaluate the margins for improvement of the execution time by means of parallelisation, we conducted a set of tests measuring the average time related to simulation turns. The tests considered a relatively low level of global pedestrian density (about 1 $^{ped}/_{m^2}$), with and without groups in the simulated pedestrian population, on a large corridor scenario (10.0×4.0 m^2). Results showed that the heaviest activity in the turn (over 85% of the simulation turn) are related to the perception and utility calculation phase, especially in the case with groups (due to additional group behavioural mechanisms). Therefore, the strategy of parallelism introduction should especially consider the perception and evaluation phases.

3 Parallel Computing in MakkSim

Parallel Sequential Simulations – The simplest option for improving execution time of a simulation campaign is associated to the parallel execution of several overall sequential simulation processes each associated to an execution core. This procedure has been useful in the phase of model calibration and validation: the exploration of the parameter ranges, as well as of the pedestrian densities of the scenario, requires the execution of a consistent number of simulations in same scenarios. Obviously this approach does not bring any speedup of a single simulation.

Regional Parallel Update – Even considering a sequential update strategy, constraining the execution of agent updates in parallel due to dependencies among actions, it might be possible to identify agents whose choices and actions do not actually influence each other. This can be done by analysing the agents arrangement in the environment and identifying independent regions: the notion of region is a (minimal) set of cells including agents whose choices can influence each other's actions. This means that the sets of agents situated in independent regions could be actually updated in parallel. MakkSim could be therefore parallelised by introducing a procedure that is able to dynamically recognise independent regions.

The higher the number of regions identified during one turn of the simulation, the higher level of parallelism will be achieved, increasing therefore the speedup. However, computational costs of the regions recognition algorithm can be significant, especially when the density of pedestrians is high. To estimate this cost we defined a potential approach: the problem of region identification can be expressed as identifying all connected components in a graph, with the variation that the edges are obtained using function that calculates the area of influence of an agent. The strategy we propose comprises two phases: a preliminary step in which the edges are calculated, saving information about reciprocal influence of agents (which can be used also to optimise the perception phase) and a second step where the obtained graph is visited with through the *Breadth First Search* algorithm, whose time complexity is $O(|V|+|E|)$. Since the graph is not oriented and the number of the edges could tend to $|V^2|$ (in case of high local density of pedestrians), the overall costs could be extremely significant. Moreover, the overall speedup will be constrained by the procedure of *density field* updating, that represents a bottleneck since the floor field is unique for the simulation and its structural integrity is fundamental. For all these reasons, we consider this option not promising.

Parallel Agents Update – By assuming a parallel agent activation strategy, the activity flow of the simulation update phase would follow a three step procedure: (i) *update of choices* and *conflicts detection*, that represents the call of the *chooseAction* function for each agent of the simulation; this function, however, now generates an intention of movement whose management will also allow for the identification of conflicts; (ii) *conflicts resolution*, that is the resolution of the detected conflicts between agent intentions, solved employing new rules in the model; (iii) *agents movement*, that is the simple update of agent positions exploiting the previous conflicts resolution, and *field update*, that is the recalculation of the density field regarding the new positions of the agents. The overall computational costs are increased, because there is the conflict resolution phase in addition. On the other hand, in the procedure we want to propose these costs linearly increase with the number of agents and, moreover, it must be noted that all of these activities can be now parallelised by using threads. In Alg. 1 the pseudo-code of this update procedure is proposed: *AgentChoices* is a table of the same size of the environment, where the current movement choices of agents are saved. By using this structure we detect collisions simultaneously to the storage of agents' choices through the *addChoice* function. The costs associated to the detection of conflicts are therefore linear with the number of agents. Conflicts resolution phase also has linear costs (considering the number of conflicts), since we adopt a method from the literature [6] that consists in executing a lottery for each conflict of the current step: to preserve a consistent environmental condition, some of the movement intentions can be canceled by the environment. As a consequence, the last activity only has to finalise the remaining movement intentions of all agents, that are now legitimate.

A proposal for the parallelisation of this update procedure is shown by the pseudo-code in Algorithm 2 (only the most significant procedures are described, for sake of space). The first function represents the main procedure and implements the parallelisation strategy by using three types of thread: the *Updater*, whose objective is to perform the update of agent choices, as well as to store

Algorithm 1. Parallel update and *addChoice* function

```
function PARALLELUPDATE
    AgentChoices ← $newTable$(Environment.rows, Environment.columns)
    for $\alpha \in$ Agents do
        Choice = $updateChoice(\alpha)$
        AgentChoices.$addChoice$(Choice)
    end for
    $solveConflicts$(AgentChoices)
    for $\alpha \in$ Agents do
        $move(\alpha)$
    end for
end function

function ADDCHOICE(Choice, Conflicts)
    if Table[Choice.x][Choice.y].$length() == 1$ then
        Conflicts.$append$((Choice.x, Choice.y))
    end if
    Table[Choice.x][Choice.y].$append$(Choice)
end function
```

Algorithm 2. threadsParallelUpdate() and Updater.run

```
function THREADSPARALLELUPDATE
    Arbiter ← $newArbiter$(countCPUs())
    AgentChoices ← $newTable$(Environment.rows, Environment.columns)
    for $i \in [1, \text{countCPUs}()]$ do
        $start$($newUpdater$(Agents, AgentChoices, Arbiter))
        $start$($newConflictSolver$(AgentChoices, Arbiter))
        $start$($newAgentMover$(Agents, Arbiter))
    end for
end function

function UPDATER.RUN
    while $length(Agents) > 0$ do
        $\alpha \leftarrow tryToExtract$(Agents)
        if $\alpha \neq \emptyset$ then
            Choice = $updateChoice(\alpha)$
            $lock$(AgentChoices)
            AgentChoices.$addChoice$(Choice)
            $unlock$(AgentChoices)
        end if
    end while
    $Arbiter.endJob$("$Updater$")
end function
```

them in the respective data structure[2]; the *ConflictSolver*, which performs the conflict resolution activity and updates the agents choices; the *AgentMover*, that

[2] Method *tryToExtract* locks the Agent list and returns its first element, by removing it from the set. If the list is empty it returns $\emptyset$, so the *Updater* has finished its work.

updates agents' positions and density field. Synchronisation between these processes is made by the *Arbiter* monitor that ensures the completion of the steps of the update procedure, and by using methods *lock* and *unlock* for synchronising shared data resources (e.g. *Agents* or *AgentsChoices*).

We expect to achieve a significant speedup by implementing this approach: in particular, while the final phases of actual movement and density field recomputation present bottlenecks in the shared resources, the *updateChoices* procedure (which is the most time expensive step in MakkSim simulation cycle) only presents a minimal race condition associated to the storage of conflicting movement intentions.

4 Conclusions

In this paper we have systematically analysed the different options for the introduction of parallel computation techniques to decrease the execution time of simulations employing the MakkSim system. The the adoption of a conceptually parallel agents' activation strategy resulted the most promising one and we are now working at changes in the MakkSim system allowing the experimentation of the described parallel computing approach and the experimental evaluation of the achieved speedup.

References

1. Bandini, S., Manzoni, S., Vizzari, G.: Situated cellular agents: a model to simulate crowding dynamics. IEICE Transactions on Information and Systems: Special Issues on Cellular Automata E87-D(3), 669–676 (2004)
2. Bandman, O.: Using multi core computers for implementing cellular automata systems. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 140–151. Springer, Heidelberg (2011)
3. Cannataro, M., Gregorio, S.D., Rongo, R., Spataro, W., Spezzano, G., Talia, D.: A parallel cellular automata environment on multicomputers for computational science. Parallel Computing 21(5), 803–823 (1995)
4. Helbing, D., Molnár, P.: Social force model for pedestrian dynamics. Phys. Rev. E 51(5), 4282–4286 (1995)
5. 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)
6. Kirchner, A., Nishinari, K., Schadschneider, A.: Friction effects and clogging in a cellular automaton model for pedestrian dynamics. Phys. Rev. E 67, 056122 (2003)
7. Klüpfel, H.: A Cellular Automaton Model for Crowd Movement and Egress Simulation. Ph.D. thesis, University Duisburg-Essen (2003)
8. 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)
9. Vizzari, G., Manenti, L., Crociani, L.: Adaptive pedestrian behaviour for the preservation of group cohesion. Complex Adaptive Systems Modeling 1(7) (2013)

CA Agents for All-to-All Communication Are Faster in the Triangulate Grid

Rolf Hoffmann[1] and Dominique Désérable[2]

[1] Technische Universität Darmstadt, FB Informatik, FG Rechnerarchitektur, Hochschulstraße 10, 64289 Darmstadt, Germany
hoffmann@ra.informatik.tu-darmstadt.de

[2] Laboratoire LGCGM UPRES EA 3913, Institut National des Sciences Appliquées, 20 Avenue des Buttes de Coësmes, 35043 Rennes, France
deserabl@insa-rennes.fr

Abstract. The objective was to find the behaviour of agents to solve the all-to-all-communication task in the cyclic triangulate and square grids in shortest time. The agents should be reliable, meaning that they are successful on almost any initial configuration. In order to solve the problem, the multi-agent system was modeled by Cellular Automata with synchronous updating, and the behavior of the agents was modeled by an embedded finite state machine (FSM). Agents can move or stay, and turn to any direction. An agent is able to leave a trace by setting a color flag on its site. Colors allow indirect communication similar to pheromones, speed up the task and contribute to a better reliability. More reliable agents could be found by using different initial control states for the agent's FSMs. A simple genetic procedure based on mutation was used to evolve near optimal FSMs for the triangulate and square grid. Agents in the triangulate grid can solve the task in around 2/3 of the time compared to agents in the square grid. The communication time depends also on the density of agents in the field, e.g. agents with density 4/(16 x 16) turned out to be the slowest.

Keywords: All-to-All communication, Cellular Automata, Multi-agent-system, Evolving Finite State Machines, Square and Triangulate Grids, Indirect Communication.

1 Introduction

Problem and Objectives. Several agents that are moving around in a Cellular Automata grid are able to solve the all-to-all communication task. Initially each one has got a mutually exclusive part of the whole information which can be distributed when the agents meet locally. The task is considered *successful* when all agents have gathered the complete information. Information parts already gathered by an agent can be handed over to another agent. The objectives here are:

- find reliable agents for the square grid (S-grid "S") and the triangulate grid (T-grid "T")
- compare the communication time in S and T for a varying density of agents.

The general goal of our project is to find systematically the local behavior of moving agents in a multi-agent system modeled by Cellular Automata in order to fulfil a given global task.

Previous and Related Work. All-to-all communication (A2A) is a very common task in distributed computing. A2A in multi-agent systems is related to multi-agent problems like finding a consensus [1], synchronizing oscillators, flocking theory or rendezvous in space [2], or in general to distributed algorithms with robots [3]. The problem's specification can depend on many fixed or dynamic varying parameters like the number and location of nodes, the number and location of processes, the number, users and properties of the communication channels and so on.

In former investigations [4] we have tried to find the best algorithms for the *Creatures' Exploration Problem*, in which the creatures (agents) have the task to visit all empty cells in shortest time. The problem herein is related to it with respect to finding an optimal movement of the agents. But this task is different: now the agents shall exchange all their individual information in shortest time.

We have already studied A2A [5–9] in the S-grid. In these investigations, environments were used with and without border (cyclic), with and without obstacles, with and without colors, and with different local communication situations. The main results were

- environments with border are easier (faster) to solve
- colors speed up the task by a factor of around 2
- time-shuffling (alternating two FSMs in time) speeds up the task
- for a field of size 33 x 33 and 16 agents, the best reached communication time was
  - 406 time steps, using actions turn right/left and move, time-shuffling two FSMs with 6 states each, no coloring [8]
  - 195 time steps, using actions turn right/left/straigth/back and move, coloring, color-FSM and action-FSM with 8 states each [9]
  - 320 time steps, using actions turn right/left/straigth and move, coloring, 6-state FSM [7].
- the evolved agents (FSMs) were not always reliable.

The former modeling and parameter setting differs in many details from the one used here. Based on this experience we defined a new, more clear modeling (especially with a von-Neumann communication) to be used for S and T grids.

Our research in general is also related to works like: evolving optimal rules for cellular automata (CA) [10, 11], finding out the center of gravity by marching pixels by evolutionary methods [12], modeling multi-agent systems in CA to simulate pedestrian movement [13], or traffic flow [14].

Contribution and Organization. The main contribution is the finding of reliable, near optimal agents controlled by a finite state machine solving the A2A task in the square and triangulate grids, and showing that agents in the triangulate grid are around 1.5 times faster than in the square grid.

Topology and basic communication properties of the square and triangular grids are given in Sect. 2. The modeling of the multi-agent system (MAS) is presented in Sect. 3. The genetic procedure used to evolve the agents' behavior is described in Sect. 4. Sect. 5 provides the results of this investigation and Sect. 6 concludes.

2 Topology of the CA Networks: S-Grid and 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) 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" in the sequel. In the right block, two additional links $(x-1, y-1)$, $(x+1, y+1)$ are provided in the diagonal NW–SE direction, giving a 6–valent torus usually denoted as "triangulate" and labeled "T" or "T-grid" in the sequel [15]. 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 $n - 1$. To be precise, let us finally note that the dual *cellular* tilings S^* and T^* as displayed in Fig. 2 and associated to S and T, are respectively the $\{4, 4\}$ square tiling and the $\{6, 3\}$ (homeomorphic) honeycomb, where $\{p, q\}$ is the Schläfli symbol.

The basic routing schemes are driven by the Manhattan distance in S and by the so-called "hexagonal" distance in T [16]. Global communications such as "One-to-All" broadcasting or "All-to-All" gossiping are frequently used in parallel applications: respectively, *one* node diffuses a message to *all* nodes (broadcasting) whereas *all* nodes diffuse their own message to *all* nodes (gossiping) [17]. For a given topology, their exists at least one deterministic protocol for each global communication. But in the context of multi-agent systems herein the context is quite different, because on one hand the number of agents is not

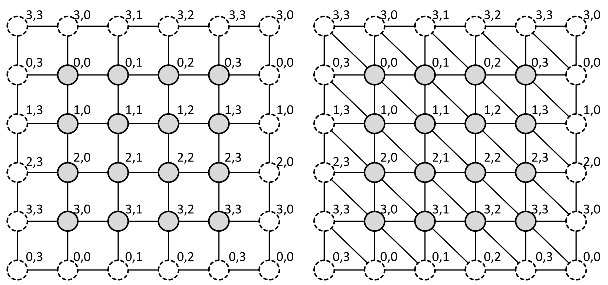


Fig. 1. Tori "S" and "T" of size $n = 2$, of order $N = 16$, labeled in the XY coordinate system; their number of links is $2N$ for S and $3N$ for T

necessarily the number of nodes and on the other hand agents' trajectory is not deterministic.

Two important parameters for the routing task in the networks are the diameter and the mean distance. 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 mean distance gives an average for the performance of the routing. Diameter D_n and mean distance $\bar{\delta}_n$ are given by [18]

$$D_n^S = \sqrt{N}\ ;\ D_n^T = \frac{2(\sqrt{N}-1)+\varepsilon_n}{3} \tag{1}$$

$$\bar{\delta}_n^S = \frac{\sqrt{N}}{2}\ ;\ \bar{\delta}_n^T \approx \frac{1}{6}\left(\frac{7\sqrt{N}}{3} - \frac{1}{\sqrt{N}}\right) \tag{2}$$

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 ratios denoted by

$$D_n^{T/S} \approx 0.666\ ;\ \bar{\delta}_n^{T/S} \approx 0.775 \tag{3}$$

between diameters and mean distances. Fig. 2 highlights the distances from an arbitrary cell, and thereby the diameters, in this family of CA networks of size 3.

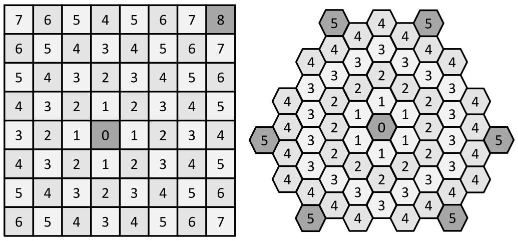


Fig. 2. Distances and antipodals from a center cell in the cellular representation of S and T for $n = 3$: $D_3^S = 8$, $\bar{\delta}_3^S = 4$; $D_3^T = 5$, $\bar{\delta}_3^T \approx 3.09$

3 CA Modeling of the Multi-agent System

The whole system is modeled by cellular automata (CA). It consists of an environment ($M \times M$ S- or T-grid, where $M = 2^n$ herein) without borders (cyclic wrap-around) and k uniform agents. We decided not to use borders because this case is more difficult to solve because the agents cannot use the borders as an orientation where they can meet. The state of CA cell is either *empty* or the state of an agent (if an agent is situated on that site).

Agent's State. The *state* of an agent is:

$$\texttt{state} = \{\texttt{IDentifier}, \texttt{Direction}, \texttt{ControlState}, \texttt{CommunicationVector}\}.$$

The agents are distinguished by their identifier $\texttt{ID} \in \{0 \ldots N_{agents} - 1\}$. Thereby the agent's trajectories can be traced, and the ID can be used for resolving conflicts and as an information to vary the initial control state. The moving direction is $\texttt{Direction} \in \{0..3\}$ for S and $\texttt{Direction} \in \{0..5\}$ for T, respectively. The `ControlState` is the state of an embedded finite state automaton, controlling the actions.

Communication Method. In order to model the distribution of information, a bit vector `CommunicationVector` of k length is stored in each agent. At the beginning the bits are set mutually exclusive ($\texttt{bit}(i)$=1 for agent(i)). Agents exchange their information when they meet in a certain communication situation. We decided that an agent can read all the information from the other agents from its nearest neighbours (4 in S and 6 in T). This communication situation does not depend on the agents moving direction, as in the communication situations we used earlier. We think that this new definition is more simple and will yield similar results. The exchange of information is modeled by simply OR-ing the communication vectors of the involved agents. The task is successfully solved when the k–vector becomes $(11 \ldots 1)$ for all agents.

Actions. An agent can perform the three basic actions `move`, `turn`, `setcolor` independently of each other.

- The agent moves in the current direction if `move`=1 when it can move (free, not blocked), otherwise it waits. The agent waits unconditionally if `move`=0.
- For the S-grid, the basic action $\texttt{turn} \in \{0, 1, 2, 3\}$ defines the new direction: $\textit{direction} \leftarrow \textit{direction} + \texttt{turn} \times 90°$.
  This means that the agent can turn to any of the four directions.
  For the T-grid, the basic action $\texttt{turn} \in \{0, 1, 3, 5\}$ defines the new direction: $\textit{direction} \leftarrow \textit{direction} + \texttt{turn} \times 60°$.
  This means that the T-agent cannot turn to $\pm 120°$. This decision was made because the cardinality of `turn` should be the same, in order to have the same complexity of abilities for the S- and T-agents.
- Apart from the agent's movement and the information exchange, an agent has indirect communication capabilities. Each cell of the environment contains a *color* (status flag) which is either 0 or 1 and used as an input for the decision making process. The color can be seen as a tracing information like a "pheromone" left by other agents or even by the reading agent itself. The agent is able to change the color of the cell on which the agent is currently located:
  color ← 0 if `setcolor`=0, and *color* ← 1 if `setcolor`=1.

Thus in total there are 16 possible actions that an agent can perform. The actions can be written in abbreviated form, using $\texttt{turn} \in \{S, R, B, L\}$ (Straight, Right, Back, Left), $\texttt{move} \in \{m, .\}$ (move, wait) and $\texttt{setcolor} \in \{0, 1\}$. Thus the action set is:

{*Sm0, Sm1, S.0, S.1, Rm0, Rm1, R.0, R.1*
Bm0, Bm1, B.0, B.1, Lm0, Lm1, L.0, L.1}.

Input Information. The information on which an agent can react on, is

– the color of the current cell the agent is situated on,
– the color of the cell ahead (also called the *front cell*),
– if there is an agent in front or not,
– if there are agents that want to move to the same front cell (conflict),
– the agent's own control state.

Conflicts. An agent cannot move if it detects an agent in front or is a looser of the conflict resolution protocol. A conflict occurs when two or more agents want to move to the same front cell (cell in conflict). In order to detect a conflict an extended neighborhood [5] is needed (Manhattan distance 2 in the moving direction). Alternatively, especially when a fast hardware implementation is pursued, the conflict detection can be performed by an arbitration logic [4] available in each cell. The arbitration logic evaluates the move requests coming from the agents and replies asynchronously by an acknowledge signal in the same clock cycle. In order to resolve a conflict, a resolution strategy has to be defined. We defined that the agent with the lowest ID has priority.

Control FSM. The decision upon which action will be performed depends on the behavior of the agent. The behavior (algorithm) of an agent is defined by a finite state machine (FSM) of type MEALY. A FSM contains a state register and a transition/output table. Input of the table is the current input x and the current state s, output is the next state s' and current output y, e.g. Fig. 3.

Input x of the concrete FSM used here is the own control state s of the FSM, the move condition $x = \texttt{canmove} \in \{0, 1\}$, the color of the own cell the agent is situated on, and the color of the front cell. The inverse move condition is called *blocked*. Thus altogether there are 8 different input values. The move condition is computed by a separate function, that evaluates to TRUE if (i) there is no agent in front and (ii) in case of conflict the own ID is the lowest compared to the others.

Output of the FSM is $y = (\texttt{move}, \texttt{turn}, \texttt{setcolor})$.

In order to keep the control automaton simple, we restrict the number of states and actions to a certain limit (see Sect. 4). To solve the problem very general either theoretical or practical with respect to all interesting parameters is too difficult. Therefore we have specialized our investigation. The grid size was set to 16×16. The number of agents was set to $k = 16$ for the genetic procedure but then varied from $k = 2$ to the maximum (number of cells).

4 The Genetic Procedure

The ultimate goal is to find the optimal behavior on average for all possible initial configurations in S and T. As we cannot test all possible configurations, we restrict our investigation to a field size of 16 x 16, with a certain number of agents

```
S-agent FSM /x = 0\  /x = 1\  /x = 2\  /x = 3\  /x = 4\  /x = 5\  /x = 6\  /x = 7\
blocked        0        1        0        1        0        1        0        1
color          0        0        1        1        0        0        1        1
frontcolor     0        0        0        0        1        1        1        1
state       0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3
|nextstate  2 3 1 1  0 3 3 2  1 3 0 2  0 0 2 1  1 2 2 0  2 3 2 0  2 2 3 0  3 1 0 2 |
|setcolor   1 1 0 0  0 1 0 1  0 0 0 1  1 0 1 1  0 0 0 0  0 0 0 1  0 0 0 1  1 0 0 0 |GENOM
|move       1 1 0 1  0 1 1 1  1 1 1 1  1 1 1 0  1 1 1 1  0 0 0 0  0 0 0 1  0 1 0 0 |S-agent
|turn       3 0 1 0  1 1 1 2  3 0 0 3  2 1 2 3  0 1 2 1  3 0 1 3  2 3 3 3  3 2 2 3 |
index i     0 1 2 3  4 5 6 7  8 91011 12131415 16171819 20212223 24252627 28293031
```

Fig. 3. A state table for an FSM controlled S-agent moving and communicating in the S-grid. The `turn` actions for the S-agent mean: *turn* $0°/90°/180°/-90°$ for (`turn` = 0,1,2,3) – This FSM represents also the best found algorithm for the S-agents

$N_{agents} \in \{2, 4, 8, 16, 32, 64, 256\}$. We will be satisfied if we can find near optimal (fast) agents, that are also reliable, e.g. which are able to solve the problem for each N_{agents} and for all initial configurations out of a set of $N_{fields} = 1003$. Thus 1000 initial configurations were randomly generated (position, direction) for each N_{agents}, plus 3, manually designed. The manually designed are difficult to be solved by simple uniform agents, because it may easily happen that agents never meet (when agents follow synchronously the same strategy). The first is a queue of N_{agents}, all agents with direction $\rightarrow$; the second is a queue of N_{agents}, all agents with direction $\leftarrow$; and in the third configuration the agents are placed in the diagonal with maximum space between them, all agents with direction $\leftarrow$.

As the search space for different behaviors is very large we are not able to check all possible behaviors by enumeration. The number of state machines which can be coded using a state table is $K = (|s||y|)^{(|s||x|)}$ where $|s|$ is the number of states, $|x|$ is the number of different input values and $|y|$ is the number of different outputs. As the search space increases exponentially we used a genetic procedure in order to find the best behavior within a reasonable computational time limit. Nevertheless the number of states and inputs has to be kept low in order to find a good solution in reasonable time.

The concatenation of the (`nextstate`, action)-pairs (s', y) for all input combinations with index i (`state` table in Fig. 3) defines the genome of one individual, a possible solution.

One population of N individuals is updated in each generation (optimization iteration). During each iteration $N/2$ offsprings are produced from the top $N/2$ individuals. The union of the current N individuals and the $N/2$ offsprings are sorted according to their fitness, duplicates are deleted and the number of individuals is reduced to the limit of N in the pool. In order not to get stuck in a local minimum and to allow a certain diversity in the gene pool, the first b individuals from the second half of the gene pool are exchanged with the last b individuals from the first half of the gene pool. We used $N = 20$ and $b = 3$, therefore the individuals 7, 8, 9 are exchanged with the individuals 10, 11, 12, where the individuals are numbered from 0 to $N - 1$.

We experimented with the classical crossover/mutation method. Then we found that mutation only gave us similar good results. So we used here only mutation. It is subject to further research which heuristic is best to evolve state machines. In previous work we used also crossover and parallel populations, but at the moment we have no far-reaching comparisons between different heuristics to evolve state machines.

An offspring is produced by modifying separately the `nextstate` action, the `setcolor` action, the `move` action, and the `turn` action for each input combination (`index` in the FSM table):

$$\begin{aligned}
&\texttt{nextstate} \leftarrow \texttt{nextstate}{+}1 \mod N_{states} \text{ with prob. } p_1\text{, otherwise unchanged,}\\
&\texttt{setcolor} \leftarrow \texttt{setcolor}{+}1 \mod N_{setcolor} \text{ with prob. } p_2\text{, otherwise unchanged,}\\
&\texttt{move} \leftarrow \texttt{move} + 1 \mod N_{move} \text{ with prob. } p_3\text{, otherwise unchanged,}\\
&\texttt{turn} \leftarrow \texttt{turn} + 1 \mod N_{turn} \text{ with prob. } p_4\text{, otherwise unchanged.}
\end{aligned}$$

We tested different probabilities, and we achieved good results with $p_1 = p_2 = p_3 = p_4 = 18\%$.

The fitness of a multi-agent system is defined as the number of steps which are necessary to distribute (all-to-all) the information, averaged over all initial configurations (positions and directions of the agents) in a certain set. As we are looking for reliable agents, the cardinality of the set has to be reasonably high in order to be relatively sure that the agents are successful for any given initial configuration. As the behavior of the whole system depends on the behavior of the agents, we search for the agents' state algorithms (FSMs) that can solve the problem with a minimum number of steps for a large number of initial configurations.

The fitness function F is evaluated by simulating the agent system. It reflects two aspects:

1. The number of agents N_{agents} which have gathered the complete information. If an agent has gathered the complete information it is *informed.* If all agents are informed, we characterize the agent system, respectively the algorithm, as *successful.* If the agents are successful on all given initial configurations then we characterize the agent system, respectively the algorithm, as *completely successful.*
2. The number of steps in the CA simulation to reach successfulness. We will call this value *communication time* t_{comm}.

The used fitness function integrates these aspects by choosing a weight W such that a dominance relation is formed:

$$F_i = W(N_{agents} - a_i) + t_{i,comm} \qquad\qquad W = 10^4$$

where a_i is the number of informed agents, and $t_{i,comm}$ is the communication time for an initial configuration i. The first term $(N_{agents} - a)$ reflects the number of agents that are not informed. It diminishes for a successful FSM and then the relation $F_i = t_{i,comm}$ holds. Note that a lower fitness value is better. The fitness F_i is computed for each simulated initial configuration i and then averaged over all initial configurations N_{fields} in the given set as

$$F = \sum F_i / N_{fields} \ .$$

The communication time depends on the size of the field, the number of agents, the algorithm (FSM), and the given field (initial configuration). For the investigated field size of 16 x 16 the expected communication time is lower than 100. During the genetic procedure a reasonable simulation time limit was used, e.g. $t_{max} = 200$.

```
T-agent FSM /x = 0\  /x = 1\  /x = 2\  /x = 3\  /x = 4\  /x = 5\  /x = 6\  /x = 7\
blocked        0        1        0        1        0        1        0        1
color          0        0        1        1        0        0        1        1
frontcolor     0        0        0        0        1        1        1        1
state       0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3  0 1 2 3
|nextstate  1 2 1 2  1 0 3 0  2 1 0 3  1 2 1 3  1 2 0 2  0 1 3 0  2 2 1 1  2 2 1 1 |
|setcolor   1 1 1 1  0 1 1 1  0 0 1 1  0 1 0 0  0 0 0 0  1 1 1 1  0 0 1 0  1 1 1 0 |GENOM
|move       1 1 1 0  1 0 0 0  1 1 1 1  0 1 1 1  1 1 1 0  1 0 0 0  1 1 1 0  1 0 1 1 |T-agent
|turn       0 0 1 0  3 2 2 2  3 0 0 1  0 0 3 3  1 0 1 2  3 3 0 1  3 0 1 3  2 0 2 3 |
```

Fig. 4. Best evolved FSM for the T-agent. The **turn** actions for the T-agent mean: *turn* $0°/60°/180°/-60°$ for (**turn** = 0,1 2,3).

The genetic procedure starts with $N = 20$ random FSMs. Usually there is no FSM in the initial population that is successful. After some generations, some successful FSMs are found. Then, after further generations, FSMs are expected to be evolved that are completely successful.

The genetic procedure was applied in the following way. Four independent optimization runs on 1003 initial configurations were performed, with field size 16 x 16 and $N_{agents} = 8$. Then the top 3 completely successful FSMs of each run (altogether 12) were also tested for $N_{agents} = 2, 4, 8, 16, 32, 256$, each on 1000 random initial configurations plus 3 extra manually designed (agents queueing in a line, agents on the diagonal). FSMs which were completely successful on all these configurations (1003 + 5 x 1003) were extracted and ranked. Then the best FSM was selected.

Former investigation showed that it is very difficult or impossible to find reliable agents which are successful on any given initial configuration, because agents can follow similar routes which are "parallel" and therefore never intersect. In general, a certain inhomogeneity (in space or in time) or even a randomness has to be introduced to break the symmetry. Some of the approaches to make the agents more reliable, are:

1. use coloring,
2. use random-like pattern of initial colors,
3. use different species (FSMs) of agents,
4. start the agents in different control states,
5. add obstacles.

We used the 4^{th} option. Experiments showed that we could not find uniform reliable agents when all FSMs started in control state 0 or 3. Recall that we use only 4 control states. But we were able to find reliable agents, when we started some of the agents in state 0 and the others in state 1. We decided then to use the initial state = 0/1 for agents with even/odd ID.

Table 1. Communication time for N_{agents} in the T-grid and S-grid in a 16 x 16 field, averaged over 1003 initial configurations. The best found T-algorithm and best found S-algorithm were used.

N_{agents}	2	4	8	16	32	256
T-grid	58.43	78.30	58.68	41.25	28.06	9.00
S-grid	82.78	116.12	90.93	63.39	42.93	15.00
T/S	0.706	0.674	0.645	0.651	0.690	0.600

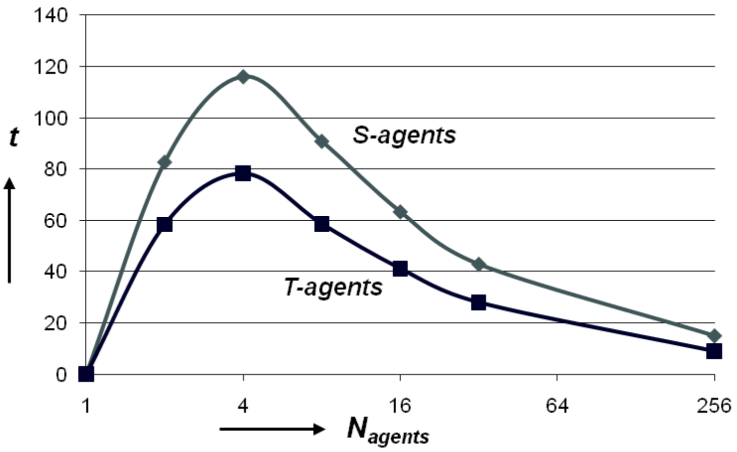


Fig. 5. Communication time for N_{agents} in the T-grid and S-grid in a 16 x 16 field. The T-agents are significantly faster than the S-agents, around 33 %. For $N_{agents} = 4$ maxima appear.

5 Comparison of the Evolved S- and T-Agents

The best found reliable FSM for the S-agent is shown in Fig. 3, and the best found T-agent is shown in Fig. 4. The communication time was evaluated by simulation for all $N_{agents} = 2, 4, 8, 16, 32, 256$, and for each case 1003 initial configurations. The agents were completely successful on all 5015 initial configurations using the same algorithm, one for the S-grid, one for the T-grid. The agents start in

```
SGRID FSM=1 FIELD=15 t=0                 t=56                              t=114
  . . . . . . . . . . . . . . . .    <1. . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .
  . . . . . . . . . . .^0 . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . .<0 . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . .v1     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . . .    . . . . . . . . . . . . .0> .v1

colors
  . . . . . . . . . . . . . . . .     . 1 . 1 . 1 . 1 . 1 . 1 . . 1 1    . 1 . 1 . 1 . 1 . 1 . 1 . . 1 .
  . . . . . . . . . . . . . . . .     . 1 . 1 . 1 . 1 . . . 1 . . . 1    . . . . . . . . . . 1 . . . 1 .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . . . 1 1    . . 1 . 1 . 1 . 1 . 1 1 . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . 1 . . .    . . . . . . . . . . . . 1 . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 . . 1 1    . . . . . . . . . . . 1 . . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . 1 . . .    . . . . . . . . . . . . 1 . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 . . 1 1    . . . . . . . . . . . 1 . . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . 1 . . .    . . . . . . . . . . . . 1 . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 . . 1 1    . . . . . . . . . . . 1 . . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . 1 . . .    . . . . . . . . . . . . 1 . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 . 1 . .    . 1 . 1 . 1 . 1 . 1 . . . . 1 .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . 1 . . .    . 1 . 1 . 1 . 1 . 1 . 1 . 1 . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 . 1 1 1    . . . . . . . . . . . 1 . 1 1 .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . 1 . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 . 1 1 1    1 1 . 1 . 1 . 1 . 1 . . . 1 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . 1 . . .    . . . . . . . . . . . 1 . 1 . .

visited
  . . . . . . . . . . . . . . . .     1 1 1 1 1 1 1 1 1 1 . 2 1 1 1 1    1 1 1 1 1 1 1 1 1 1 . 2 1 1 1 3
  . . . . . . . . . . . 1 . . . .     1 1 1 1 1 1 1 1 1 2 2 5 6 3 2 2    2 3 2 3 2 3 2 3 2 3 3 7 7 4 3 3
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 2 2 . 1 1    1 1 1 1 1 1 1 1 1 1 2 2 2 . 2 3
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 2 . 1 1    . . . . . . . . . . . 1 2 . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 . 1 1    . . . . . . . . . . . 1 1 . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 . 1 1    . . . . . . . . . . . 1 1 . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 . 1 1    . . . . . . . . . . . 1 1 . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 . 1 1    . . . . . . . . . . . 1 1 . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 . 1 1    . . . . . . . . . . . 1 1 . 1 1
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 . 1 1    . . . . . . . . . . . 1 1 . 1 1
  . . . . . . . . . . . . . . . 1     . . . . . . . . . . . 1 1 1 2 2    1 1 1 1 1 1 1 1 1 1 1 2 1 2 3 5
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 5 4 6 3 5
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 1 1 1    . . . . . . . . . . . 1 2 1 1 2
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 1 1 1    . . . . . . . . . . . 1 2 1 1 2
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . 1 1 1 1 1    1 2 1 1 1 1 1 1 1 1 1 2 2 1 1 3
  . . . . . . . . . . . . . . . .     . . . . . . . . . . . . 1 1 1 1    . . . . . . . . . . . 2 3 3 3 6
```

Fig. 6. Simulation of two agents in a 16 x 16 S-grid for a special initial configuration. The behaviour is defined by the best found FSM. Colors are set and reset (depicted in the middle): this information helps the agents to find each other faster. The agents build streets on which they travel more frequently (visited cells, depicted on the bottom). For this initial configuration, the S-agents need 114 time steps.

the initial control state `ID` mod 2; thereby the agents are very reliable. But we could not prove that these state machines will be successful for any arbitrary initial configuration.

Table 1 and Fig. 5 show the average communication time. It is interesting to observe that maxima were found for $N_{agents} = 4$, e.g. 4 agents communicate slower than 2 and 8 agents. Comparing T/S-agents, the ratio of communication time lies between 0.71 and 0.6, meaning that the T-agents are significantly faster. We expected a ratio of around 0.666, according to the diameter ratio $D^{T/S}$ in (3). All cases come close to this expected ratio. Note that the communication times in T and S are not related to the mean distance ratio $\bar{\delta}_n^{T/S}$.

```
TGRID FSM=2 FIELD=15 t=0                     t=13                                 t=44
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . ^0. . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . ^0. . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . .  v1    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . )1   . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . .  v0 . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . .  ^1 . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .

colors
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 . . . 1    1 . . 1 1 . . . 1 . . . . . . 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 . . . 1    . 1 1 1 . 1 . . 1 . . 1 . . . 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . 1 1 1    . . . . . . 1 1 1 . . . . 1 1 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 1 1 . .    . . 1 1 1 . . . . . . 1 1 1 . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 . . . .    . . 1 . . 1 . 1 1 1 . 1 . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 . . . .    . . 1 . . . 1 1 . . 1 1 . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . 1 . . 1    . . . 1 . . . 1 . . . 1 1 . . 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . 1 1 1    1 . . . 1 1 . 1 1 . . 1 . 1 1 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 . . . 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 1 1 . .    . . 1 1 . 1 1 . . . 1 1 1 1 . 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 . . 1 .    . . . 1 . . . 1 . . . 1 . . 1 1

visited
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 2 . . . 1    1 . . 1 1 . . . 1 . . 2 . . . 1
  . . . . . . . . . . . 1 . . . .    . . . . . . . . . . . 1 1 . . 1    . 1 1 1 . 1 . . 1 . . 1 1 . . 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . 1 1 1    . . . . . . 1 1 1 . . . . 1 1 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 1 1 . .    . . 1 1 1 . . . . . . 1 1 1 . .
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 . . 1 .    . . 1 . . 1 . 1 1 1 . 1 . . 1 .
  . . . . . . . . . . . . . . . 1    . . . . . . . . . . . 1 . . . 2    . . 1 . . . 1 1 . . 1 1 . . . 2
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . 1 . . 2    . . . 1 . . 1 1 . . . 1 1 . . 2
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . 1 1 1    1 . . . 1 2 2 1 2 . . 1 . 1 1 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . . . . . .    1 2 . . . 1 . . . 2 1 1 . . . 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 1 1 . .    . . 2 1 2 2 1 . . . 1 1 1 1 . 1
  . . . . . . . . . . . . . . . .    . . . . . . . . . . . 1 . . 1 .    . . . 1 1 . . 1 . . . 2 . . 1 1
```

Fig. 7. Simulation of two agents in a 16 x 16 T-grid for a special initial configuration. The behaviour is defined by the best found FSM. Colors are set and reset (depicted in the middle): this information helps the agents to find each other faster. The agents build honeycomb-like networks on which they travel more frequently (visited cells, depicted on the bottom). For this initial configuration, the T-agents need only 44 time steps compared to 114 time steps for the S-agents.

In the special case $N_{agents} = 256$, the system is fully packed with agents that cannot move, only communicate. Then the communication time is the diameter (1) (the communication after the initial placement is not counted).

Two sample simulations show how the agents move and set colors in order to communicate. In the S-grid of Fig. 6, the agents build orthogonal communication streets (where the agents prefer to move) by the help of the colors. We can observe a few streets at time 56 (horizontal or vertical) and more at the end (t = 114).

For the T-agents in Fig. 7, we observe that they can build honeycomb-like networks! At time 13 we observe two honeycombs and at the end (t = 44) there are several of them. It has to be mentioned that the T-agents are faster on average, but for some initial configurations they can be slower.

In a previous work [9], 195 time steps were reached for a 33 x 33 S-grid with 16 agents. For comparison, our best 12 agents, evolved for a 16 x 16 grid with 8 agents, were tested on 1003 randomly generated fields of size 33 x 33 with 16 agents. The best S-agent needed 229 time steps and the best T-agent needed 181 time steps, and the agents were reliable. Again the T-agent is faster than the S-agent. However, our T-agents are not so fast as the ones evolved in [9]. The reasons are: (1) we used only one FSM with 4 states, instead of using two FSMs with 8 states each, (2) we did not specifically evolve our agents for the field size of 33 x 33, and (3) our agents were specifically evolved for a high reliability (agents start in different initial control states).

6 Conclusion

The multi-agent system needed for simulation and optimization was described by Cellular Automata, and the agent's behavior was modeled by a finite state machine (FSM). For the triangulate and square grid, near optimal FSMs were evolved by a genetic procedure. In order to make the agents more reliable (successful on any initial configuration), half of the agents start in state 0, the other half in state 1. Thereby the agents could solve the task successfully for a large number (5015, each for the T- and S-grid) of initial configurations. T-agents can solve the task in about 2/3 of the time the S-agents needed. For the 16 x 16 grid, two and more than 4 agents can communicate faster than 4 agents. In further work it could be studied how fast and reliable agents are when using more states, more colors, obstacles, or borders.

References

1. Olfati-Saber, R., Fax, J.A., Murray, R.M.: Consensus and cooperation in networked multi-agent systems. Proc. IEEE 95(1), 215–233 (2007)
2. Lin, J., Morse, A.S., Anderson, B.D.O.: The Multi-Agent Rendezvous Problem. An Extended Summary. In: Kumar, V., Leonard, N., Stephen Morse, A. (eds.) Cooperative Control. LNCIS, vol. 309, pp. 257–289. Springer, Heidelberg (2005)
3. Santoro, N.: Distributed Algorithms for Autonomous Mobile Robots. In: Navarro, G., Bertossi, L., Kohayakawa, Y. (eds.) TCS 2006. IFIP, vol. 209, p. 11. Springer, Boston (2006)
4. Halbach, M., Hoffmann, R., Both, L.: Optimal 6-state algorithms for the behavior of several moving creatures. In: El Yacoubi, S., Chopard, B., Bandini, S. (eds.) ACRI 2006. LNCS, vol. 4173, pp. 571–581. Springer, Heidelberg (2006)
5. Ediger, P., Hoffmann, R.: Optimizing the creature's rule for all-to-all communication. In: EPSRC Workshop Automata-2008, Theory and Applications of Cellular Automata, Bristol, UK, pp. 398–412 (2008)
6. Hoffmann, R., Ediger, P.: Evolving multi-creature systems for all-to-all communication. In: Umeo, H., Morishita, S., Nishinari, K., Komatsuzaki, T., Bandini, S. (eds.) ACRI 2008. LNCS, vol. 5191, pp. 550–554. Springer, Heidelberg (2008)
7. Ediger, P., Hoffmann, R.: Solving All-to-All Communication with CA Agents More Effectively with Flags. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 182–193. Springer, Heidelberg (2009)

8. Ediger, P., Hoffmann, R.: Evolving hybrid time-shuffled behavior of agents. In: 13th Workshop on Nature Inspired Distributed Computing NIDISC, Proc. Par. Dist. Proc. IPDPS, pp. 1–8 (2010)
9. Ediger, P., Hoffmann, R.: All-to-all communication with CA agents by active coloring and acknowledging. In: Bandini, S., Manzoni, S., Umeo, H., Vizzari, G. (eds.) ACRI 2010. LNCS, vol. 6350, pp. 24–34. Springer, Heidelberg (2010)
10. Sipper, M.: Evolution of Parallel Cellular Machines. LNCS, vol. 1194. Springer, Heidelberg (1997)
11. Sipper, M., Tomassini, M.: Computation in artificially evolved, non-uniform cellular automata. Theor. Comput. Sci. 217(1), 81–98 (1999)
12. Komann, M., Mainka, A., Fey, D.: Comparison of evolving uniform, non-uniform cellular automaton, and genetic programming for centroid detection with hardware agents. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 432–441. Springer, Heidelberg (2007)
13. Dijkstra, J., Jessurun, J., Timmermans, H.: A multi-agent cellular automata model of pedestrian movement. In: Schreckenberg, M., Sharma, S.D. (eds.) Pedestrian and Evacuation Dynamics, pp. 173–181. Springer (2001)
14. Nagel, K., Schreckenberg, M.: A cellular automaton model for freeway traffic. J. Phys. 2, 2221–2229 (1992)
15. Désérable, D.: A Family of Cayley Graphs on the Hexavalent Grid. Discrete Applied Mathematics 93(2-3), 169–189 (1999)
16. 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)
17. Fraigniaud, P., Lazard, E.: Methods and problems of communication in usual networks. Discrete Applied Mathematics 53(1-3), 79–133 (1994)
18. Ediger, P., Hoffmann, R., Désérable, D.: Routing in the Triangular Grid with Evolved Agents. J. Cellular Automata 7(1), 47–65 (2012)

Parallel Implementation of Totalistic Cellular Automata Model of Stable Patterns Formation*

Anastasiya Kireeva

ICM&MG SB RAS, Pr. Lavrentjeva, 6, Novosibirsk, Russia
kireeva@ssd.sscc.ru

Abstract. Totalistic cellular automata (TCA) with weighted templates simulating pattern formation process are investigated. The investigation aims to create a method for porous media morphology synthesis according to a given set of properties such as porosity, percolation, density, etc. The proposed method is based on a parallel composition of TCA and an asynchronous CA (CA of second layer), whose evolution allows to obtain a set of patterns representing different porous media morphologies. Implementation of three-dimensional version of the CA-model is performed by means of block-synchronous transformation. Plausibility of the transformation is shown by comparison of simulation results. In addition, estimates of TCA with second layer parallel implementation efficiency is presented.

Keywords: totalistic cellular automata, parallel composition, block-synchronous mode, parallel implementation, activator, inhibitor, pattern formation, porous material.

1 Introduction

The majority of physical and biological systems are nonlinear complex systems. In such systems the self-organization phenomena arise under nonequilibrium conditions, exhibiting a self-ordering of system's elements and emergence of various spatio-temporal patterns. Analysis of stable spatial patterns and the detection of general dynamic properties of non-equilibrium systems is the key to understanding self-organization phenomena. Studying the stable patterns arising in self-organizing complex systems is important both from fundamental and practical points of view.

Investigation of stable patterns which arise in nonequilibrium dissipative systems was started by A.Turing [1], who founded the general theory of dissipative structures. A great contribution to study of self-organization problems was made by A.M.Zhabotinsky [2], B.P.Belousov, D.Young [3], V.K.Vanag [4] and others. Conventional approaches to research of complex systems self-organization, in

* Supported by (1) Presidium of Russian Academy of Sciences, Basic Research Program N 15.9-5 (2012), (2) Grant RFBR 11-01-00567a, (3) Siberian Branch of Russian Academy of Sciences, SBRAS Interdisciplinary Project 47.

general, are based on solving of nonlinear partial differential equations, which sometimes is a stiff problem. Meanwhile, discrete models, like cellular automata (CA), also are used for complex systems study. S.Wolfram showed that the phenomena of self-organization may arise in CA-models [5, 6]. In [3] D.Young presented the activator-inhibitor CA-model of spots formation on the animals' skins, based on the reaction-diffusion model of Turing [1]. The impact of the activator and inhibitor values and initial conditions on the totalistic CA evolution is investigated by L.Chua in [7] in more detail. Totalistic CA (TCA) is a class of CA, whose transition function is a threshold one of the sum of weighted values of neighboring cells' states. Application of pattern formation by TCA for obtaining porous media morphology is proposed in [8].

Since CA are intended to describe the system dynamics at micro-level, CA-simulation requires considerable computational capability and efficient parallel algorithms are needed. Achieving high performance for parallel implementation of asynchronous CA (ACA) is a difficult task. The solution of this problem is offered in [9, 10], where ACA parallelization is performed by transformation into the block-synchronous CA (BSCA). Nevertheless, in [11] it is shown that there are CA-models, in which the transformation of ACA to BSCA leads to significant changes of its evolution. Hence, for each task it is necessary to prove the plausibility of ACA to BSCA transformation.

In this paper basic properties of TCA evolution and forming stable patterns are analyzed. Based on these results a parallel composition of TCA and ACA for obtaining patterns, representing a porous media, is presented. Parallel implementation of a three-dimensional version of TCA is made using block-synchronous transformation. Analysis and verification of plausibility of ACA into BSCA transformation is performed.

The paper contains of three sections. In the first section a formal definition of TCA-model is given and CA-simulation results are presented. The second section comprises a definition of TCA and ACA parallel composition and patterns formed by its evolution in two- and three-dimensional case. The third section is dedicated to parallel implementation of the TCA and ACA composition using block-synchronous transformation. In this section verification of block-synchronous transformation applicability is performed and efficiency estimations of BSCA parallel implementation are given.

2 Totalistic CA-Model of Stable Patterns Formation

2.1 The Formal Definition of Totalistic CA-Model

Totalistic cellular automata (TCA) model is defined by the following notions [12, 9, 5]:

$$\aleph_{TCA} = \langle A, X^d, \Theta, \mu \rangle. \tag{1}$$

Here, $A = \{0, 1\}$ is a *cells state alphabet*, X^d is a *set of names*, which is finite subset of d-dimensional discrete space, it is represented by a set of coordinates:

$X^d = \{\mathbf{x}\}$, $\mathbf{x} = (i, j)$ if $d = 2$, and $\mathbf{x} = (i, j, k)$ if $d = 3$, $i = 1 \ldots M_i$, $j = 1 \ldots M_j$, $k = 1 \ldots M_k$. Θ is a *local operator* given below. *Operating mode* is denoted as $\mu \in \{\sigma, \alpha, \beta\}$, where σ stands for synchronous mode, α - for asynchronous and β - for block-synchronous mode, respectively.

A *cell* is a pair $(u, \mathbf{x})$, where $u \in A$ is a state of a cell, $\mathbf{x} \in X^d$ is a name of a cell. On the set of names the *template* $T(\mathbf{x})$, defining the nearest neighbors of each cell $\mathbf{x}$, is introduced. Hereinafter, the template is a square formed by $K \times K$ cells for $d = 2$ and a cube of $K \times K \times K$ cells for $d = 3$.

$$\begin{aligned} &T_k(\mathbf{x}) = \{\mathbf{x}, \mathbf{x} + \mathbf{a}\}, \quad \mathbf{a} = (a, b) \text{ if } d = 2; \quad \mathbf{a} = (a, b, c) \text{ if } d = 3 \\ &a, b, c = -\lfloor K/2 \rfloor, \ldots, 0, \ldots, \lfloor K/2 \rfloor \ . \end{aligned} \tag{2}$$

A *weight matrix* W_K of the same size K having positive and negative entries is defined. The positive entries are called *activators* (p). They are responsible for the growth of patterns. The negative entries are *inhibitors* (n), they impede the pattern growth. The presence of activators and inhibitors in a system allows supporting the process in an equilibrium state, and provides conditions for the stable patterns formation.

Local operator $\Theta(x)$ calculates a new state value of a cell x depending on the weighted sum of the neighboring cells defined by the template T_K:

$$\Theta(\mathbf{x}) : \ (u, \mathbf{x}) \to (u', \mathbf{x}) \ \text{ where } u' = \begin{cases} 0, \text{ if } & \sum\limits_{k=0,\ldots,|T_K|} w_k \cdot u_k \leq B \\ 1, \text{ if } & \sum\limits_{k=0,\ldots,|T_K|} w_k \cdot u_k > B \end{cases} \tag{3}$$

where w_k are entries of the weight matrix W_K, $B \in R$ is the threshold value, further in the computing experiment $B = 0$.

Application of the local operator $\Theta(\mathbf{x})$ to all cells $\mathbf{x} \in X^d$ is named as an *iteration*. It transfers the cellular array from one global state to another $\Omega(t) \to \Omega(t+1)$, where t is an iteration number. The sequence $\Sigma(\Omega) = \Omega(0), \ldots, \Omega(t), \ldots, \Omega(t_{fin})$ obtained as a result of iterative functioning of CA is named *evolution*, $\Omega(t)$ is a global state of the cellular array on the t-th iteration [9].

Further, the evolution of TCA is studied for three modes of the local operator application to the cells of CA with synchronous, asynchronous and block-synchronous operating modes. In the synchronous mode, a local operator is applied to all cells of CA, all being updated simultaneously. The asynchronous mode of CA prescribes the local operator to be applied to a randomly chosen cells, changing its state immediately. The block-synchronous mode is used for achieving high performance of ACA parallel implementation, this mode is transformation of asynchronous one, whose algorithm is described in Sect. 3.

2.2 Stable Patterns Formed by TCA Evolution

In computing experiments it is found that the TCA evolves to the two types of steady states:

- first, global state of the cellular array does not change after a certain number of iterations (t'), that is $\exists\, t' : \forall t > t' \quad \Omega(t) = \Omega(t')$,
- second, alternation of two global states occurs after a certain number of iterations (t'), i.e. $\exists\ t' : \Omega(t' + 2 \cdot k) = \Omega(t')$ and $\Omega(t' + 2 \cdot k + 1) = \Omega(t' + 1),\ k \in N$.

The main parameters affecting the TCA evolution are the template of size K, the initial state of the cellular array $\Omega(0)$ and the values of weight matrix entries $w_i \in W_K$.

The template size is taken equal to $K = 7$.

Initial states of cellular array $\Omega(0)$ are selected as follows:

- Ω_1 is a global state containing one nucleation cell in the center of the cellular array, other elements of the array being equal to zero. A nucleation cell is a cell with the state $u = 1$. Simulation with such a $\Omega(0)$ is interesting because the concept of the nucleation cell is widely used in different sciences. For example, in physics the crystals are formed from nucleus of silicon carbide, in chemistry nano-clusters are created from one nucleus and in biology colonies of bacteria grow from several randomly allocated nucleation cells.
- Ω_2 is a regular distribution of cells in state $u = 1$ with a given probability P_{Ω_2},
- Ω_3 is a irregular distribution of nucleation cells with probabilities $P^k_{\Omega_3}, k \in N$ depending on the cells' coordinates.

Two types of weight matrix are used in the experiments:

1. weight matrix containing two different values of entries $n < 0$ and $p > 0$:

$$\begin{aligned}
&d = 2 : w_{kl} = \begin{cases} p, \text{ if } |k| < \lfloor K/2 \rfloor \quad \& \quad |l| < \lfloor K/2 \rfloor \\ n, \text{ otherwise} \end{cases} \\
&d = 3 : w_{klm} = \begin{cases} p, \text{ if } |k| < \lfloor K/2 \rfloor \quad \& \quad |l| < \lfloor K/2 \rfloor \quad \& \quad |m| < \lfloor K/2 \rfloor \\ n, \text{ otherwise} \end{cases} \\
&k, l, m = -\lfloor K/2 \rfloor, \ldots, 0, \ldots, \lfloor K/2 \rfloor.
\end{aligned} \tag{4}$$

2. weight matrix containing three different values of entries $n < 0$, $n_1 < 0$ and $p > 0$:

$$\begin{aligned}
&d = 2 : w_{kl} = \begin{cases} p, \text{ if } |k| < (\lfloor K/2 \rfloor - 1) \quad \& \quad |l| < (\lfloor K/2 \rfloor - 1) \\ n_1, \text{ if } |k| = (\lfloor K/2 \rfloor - 1) \quad \& \quad |l| = (\lfloor K/2 \rfloor - 1) \\ n, \text{ otherwise} \end{cases} \\
&d = 3 : w_{klm} = \begin{cases} p, \text{ if } |k| < \lfloor K/2 \rfloor - 1 \;\&\; |l| < \lfloor K/2 \rfloor - 1 \;\&\; |m| < \lfloor K/2 \rfloor - 1 \\ n_1, \text{ if } |k| = \lfloor K/2 \rfloor - 1 \;\&\; |l| = \lfloor K/2 \rfloor - 1 \;\&\; |m| = \lfloor K/2 \rfloor - 1 \\ n, \text{ otherwise} \end{cases} \\
&k, l, m = -\lfloor K/2 \rfloor, \ldots, 0, \ldots, \lfloor K/2 \rfloor.
\end{aligned} \tag{5}$$

Variation of values of weight matrix entries w_i enables to get many different stable patterns. In spite of simple structure of matrix weight (4), TCA evolution with such matrix and initial state Ω_1 produces various fancy figures, geometric figures, stripes and spots spreading to the whole cellular array (Fig. 1). In [13] the theorem is formulated and proved that TCA evolution with weight matrix (4) for a specified initial state and cellular array size is uniquely determined by the ratio p/n. This theorem is true in three-dimensional case too.

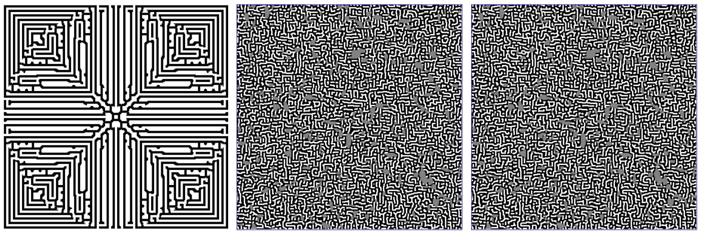


Fig. 1. Stable patterns are formed as a result of TCA evolution with $|X^2| = 500 \times 500$, $\Omega(0) = \Omega_1$ for a) $\mu = \sigma$, $n = -1$, $p = 0.89$; b) $\mu = \alpha$, $n = -1$, $p = 0.7$, c) $\mu = \alpha$, $n = -1$, $p = 1.1$

For graphical representation of global state of cellular array, cells with state $u = 1$ are marked with black, cells with state $u = 0$ - with white. A qualitative feature of TCA evolution is *motif*. The motif is an image formed by specific, repeated elements. For example, in Fig. 1a the motif is a fancy figure reminiscent of snowflake growing from the array center, and in Fig. 1b the motif is small black stripes and points and elongated gray spots.

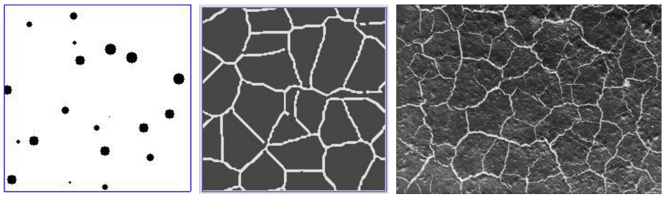


Fig. 2. Motif is obtained by TCA evolution for $|X^2| = 200 \times 200$, $\mu = \sigma$, $n = -0.5$, $n_1 = -0.1$, $p = 2$: a) $\Omega(0)$, b) $\Omega(94)$; and c) image of dry soil surface (http://www.123rf.com/photo-6872051-closeup-shot-of-dry-eroded-soil-texture.html)

Using a larger number of different values of the weight matrix entries enables to obtain more intricate motifs, similar to the observed in the nature. For example, patterns formed by TCA evolution with weight matrix (5) resemble images

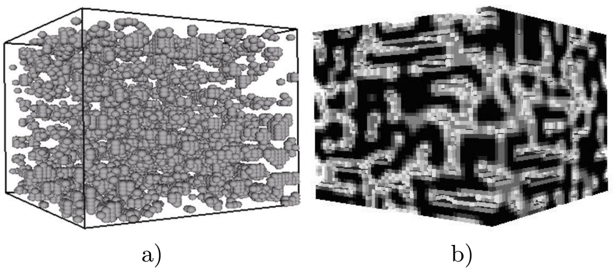


Fig. 3. Motifs are obtained by TCA evolution for $|X^3| = 50 \times 50 \times 50$, $\mu = \sigma$, $n = -0.5$, $n_1 = -0.1$ and a) $p = 2$, $\Omega(0) = \Omega_2$ with $P_{\Omega_2} = 0.005$, b) $p = 5$, $\Omega(0) = \Omega_2$ with $P_{\Omega_2} = 0.01$

of a soil slice in two-dimensional case (Fig. 2). As well, in three-dimensional case TCA evolution generates different porous materials for different values of activators and inhibitors (Fig. 3). Such motifs can be used as models for research of porous medium properties.

3 Totalistic CA Model with Second Layer

For the formation of a heterogeneous porous media a parallel composition of two cellular automata: totalistic CA ($\aleph_{TCA}$) and CA of second layer ($\aleph_{2L}$), is used. Totalistic CA with a second layer allows to take into account environment heterogeneity, being able to change dynamically the weight values depending on environment's parameters. It gives the opportunity to form stable patterns similar to real motifs arising in nature.

CA of the second layer simulates the external environment which affects the patterns formation dynamics. For that CA modeling the following process may be used: heat exchange, reagents distribution over the surface, phase separation, change in the chemical properties.

Totalistic CA and second layer CA form a parallel composition as follows [14]:

$$\aleph_S = \Upsilon(\aleph_{TCA}, \aleph_{2L}) \tag{6}$$

Between cell names of two CA there exists one-to-one correspondence: $X_{TCA} = \gamma(X_{2L})$.

$\aleph_{TCA} = \langle A, X^d, \Theta', \mu \rangle$ is similar to (1) with the exception of local operator $\Theta'(\mathbf{x})$, which calculates new state values like in (3), using weight coefficients depending on cells states of $\aleph_{2L}$:

$$\Theta'(\mathbf{x}) : (u, \mathbf{x}) \to (u', \mathbf{x}) \text{ where } u' = \begin{cases} 0, \text{ if } \sum\limits_{k=0,\ldots,|T_7|} f(v_k) \cdot u_k \le B \\ 1, \text{ if } \sum\limits_{k=0,\ldots,|T_7|} f(v_k) \cdot u_k > B \end{cases} \tag{7}$$

where v_k is the state of the k-th cell in the template $T_7(\mathbf{x}) \subset X_{2L}$.

Two functions $f(v_k)$ are further used:

$$f_1(v_k) = \begin{cases} \langle v \rangle, \text{ if } w_k > 0 & (\textit{activator}) \\ w_k, \text{ if } w_k \leq 0 & (\textit{inhibitor}) \end{cases} \text{ where } \langle v \rangle = \frac{\sum_{l=0,\ldots,|T_7|} v_l}{|T_7|} \tag{8}$$

and

$$f_2(v_k) = \begin{cases} \dfrac{N(a) - N(b)}{|T_7|}, \text{ if } w_k > 0 & (\textit{activator}) \\ w_k, \text{ if } w_k \leq 0 & (\textit{inhibitor}) \end{cases} \tag{9}$$

where $N(a)$ and $N(b)$ are numbers of cells $\mathbf{x} \in X_{2L}$ with state "a" and "b", respectively, associated with the template $T_7(\mathbf{x})$.

CA model simulating fire propagation is chosen as the second layer:

$$\aleph_{2L} = \langle A_{2L}, X_{2L}^d, \Theta_{2L}, \alpha \rangle. \tag{10}$$

The state alphabet is $A_{2L} = \{0, 1, 2\}$, where 0 denotes the clear ground, 1 is tree and 2 denotes fire. The set of names of $\aleph_{2L}$ is isomorphic to the set of names of $\aleph_{TCA}$: $X_{2L}^2 = X^2$ and $X_{2L}^3 = X^3$. The operating mode is asynchronous. The local operator $\Theta_{2L}(\mathbf{x})$ calculates new state of cell $\mathbf{x} \in X_{2L}$ depending on the neighboring cells' states associated with the template $T_{2L}(\mathbf{x}) = \{(\mathbf{x} + \mathbf{a})\}$, $\mathbf{a} \in \{(-1, 0), (0, 1)(1, 0)(0, -1)\}$ in 2d-case, and $\mathbf{a} \in \{(-1, 0, 0), (1, 0, 0,)(0, -1, 0)$ $(0, 1, 0), (0, 0, -1), (0, 0, 1)\}$ in 3d-case, here $\mathbf{a}$ is a shift vector.

$$\Theta_{2L}(\mathbf{x}): \ (v, \mathbf{x}) \to (v', \mathbf{x})$$
$$\text{where } v' = \begin{cases} 0, \text{ if } v = 2 \\ 1, \text{ if } v = 0 \quad \& \quad rand < P_t \\ 2, \text{ if } v = 1 \quad \& \quad (\exists\, (v_k, x_k): \ v_k = 2, \ x_k \in X_{2L}) \quad \& \quad rand < P_f \end{cases} \tag{11}$$

where (v_k, x_k) is one of the neighboring cells in $T_{2L}(\mathbf{x})$, $rand \in [0, 1]$ is a random number from a real interval in set $\mathbb{R}$, P_f and P_t are the probabilities of self-ignition and growth of tree, respectively.

A variety of motifs similar to the patterns, emerging in different natural phenomena, arises for different types of the function $f(v_k)$. For example, Fig. 4a shows a motif similar to bacteria colony which is formed by $\aleph_S$ with the weight matrix (5), $\Omega(0) = \Omega_2, (P_{\Omega_2} = 0.001)$ and $f(v_k)$ calculated by (8). Also, Fig. 4c represents a motif similar to a porous material morphology, the motif being obtained by $\aleph_S$ with the same parameters as in previous case but for $f(v_k)$ calculated by (9). There are motifs very much resembling porous materials in the three-dimensional case too (Fig. 4e).

TCA with dynamically changing weight coefficients ($\aleph_S$) can be used for study and synthesis of porous materials properties. Synthesis of a heterogeneous porous medium with complex pores arrangement is a difficult task. The idea of using CA for computer representation of porous material morphology and calculation of its characteristics is presented in [8]. During the CA evolution after each

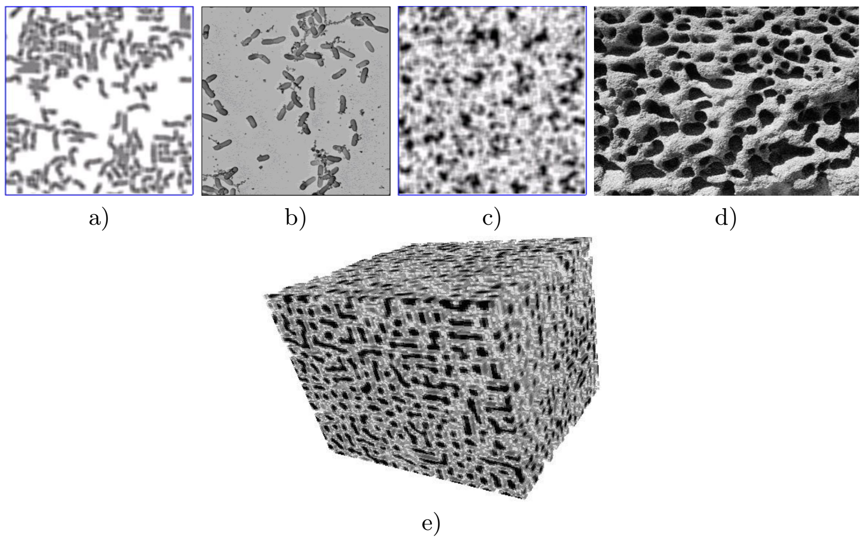


Fig. 4. Motifs are formed during the asynchronous $\aleph_S$ evolution and images of real natural patterns: a) the state of $\aleph_S$ with $|X^2| = 200 \times 200$, $n = -0.5$, $n_1 = -0.1$, $p = 1.2$, $\Omega(0) = \Omega_2$, $(P_{\Omega_2} = 0.001)$ and $P_f = 0.0001$, $P_t = 0.01$, b) the photograph of bacteria colony, c) the state of $\aleph_S$ with $|X^2| = 200 \times 200$, $n = -0.5$, $n_1 = -0.1$, $p = 3$ and $P_f = 0.01$, $P_t = 0.7$, d) porous material made from recyclable materials "Shahosintetik" (the photograph is taken from the site http://www.potram.ru/index.php?page=113), e) the state of $\aleph_S$ with $|X^3| = 100 \times 100 \times 100$, $n = -0.5$, $n_1 = -0.1$, $p = 2$ and $P_f = 0.5$, $P_t = 0.00005$

iteration the porous medium characteristics such as percolation degree, porosity, channels tortuosity, the number of connected components, are calculated. These characteristics are used for the analysis and selection of materials with the specified morphology. Fig. 5 shows the global states of $\aleph_S$ evolution in the two-dimensional and three-dimensional cases. It is seen that porous medium morphology is changing during the CA evolution: the medium porosity decreases and the density increases. Observing the process during CA evolution one can choose material with necessary properties.

Inhomogeneous porous materials study is an urgent problem in oil and gas deposits research. It is most important, because the study of morphology and characteristics of the deposits facilitates the detection of rock properties favorable for oil and gas accumulation, and moreover, full deposit development. As is known, oil and gas reservoirs are sandstone and limestone rocks, characterized by a high degree of porosity and percolation, the large number of cracks and caverns. In addition, the structure and physical properties of oil and gas reservoirs change along horizontal and vertical directions. Sometimes, sandstone and sand stratum change into mudstone without any regularity. TCA with second layer are capable to construct the rocks with different porosity and percolation. Moreover, TCA is able to create random arrangement of strata with various

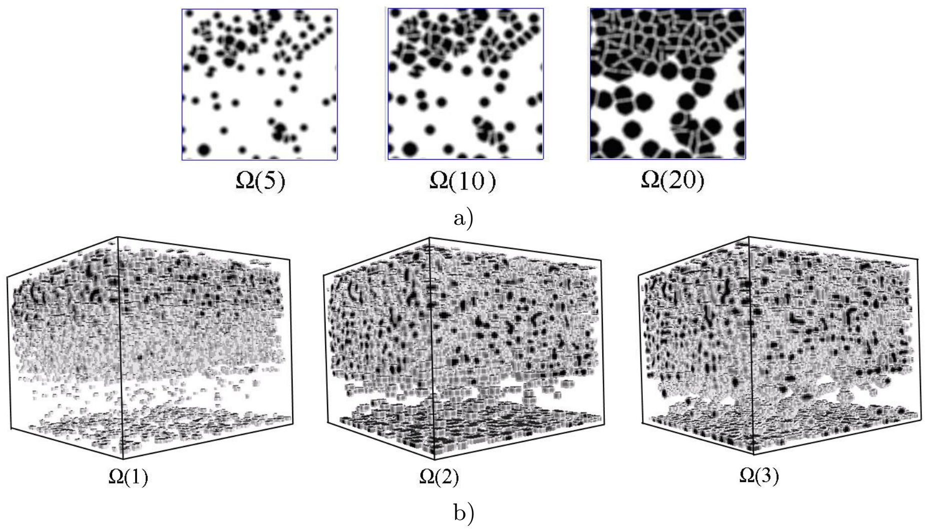


Fig. 5. The porous medium morphology changing during the CA evolution: a) motifs formed by synchronous $\aleph_S$ for $|X^2| = 200 \times 200,\ n = -0.5,\ n_1 = -0.1,\ p = 1.9$; b) motifs formed by synchronous $\aleph_S$ for $|X^3| = 100 \times 100 \times 100,\ n = -0.5,\ n_1 = -0.1,\ p = 2$

morphology. But TCA synthesis of real large-scale rocks' morphology requires a huge computational power. Therefore, it is necessary to develop efficient parallel algorithms.

4 Parallel Implementation of Totalistic CA Model with Second Layer

4.1 Transformation of Asynchronous Mode into Block-Synchronous One

The difficulties of efficient ACA parallel implementation are connected with the fact that, due to random cells choice and immediate state updating, data exchange must be performed after each boundary cell change. The problem solution is proposed in [9, 10], where ACA parallelization is performed after its transformation to the block-synchronous CA (BSCA), which is performed as follows [9].

1. On the naming set X the block B is defined. To satisfy a correctness condition the block should include the template:

$$B(x) \subseteq T(x). \tag{12}$$

   In addition, the size of the cellular array $|X|$ should be divisible by the block size $|B|$.

2. On the naming set X the partition $\Pi = \{\Pi_1, \Pi_2, ..., \Pi_{|B|}\}$ is defined as follows:

$$|\Pi_k| = |X|/|B|, \qquad \bigcup_{i=1}^{|\Pi_k|} B^i = X, \qquad B^i \bigcap B^j = \emptyset,$$
$$\Pi_k = \{x_k^i\}, \quad x_k^i \in B_k, \quad \forall\, i,j \in \{1,\ldots,|\Pi_k|\},\ i \neq j. \tag{13}$$

3. An iteration is divided into $|B|$ steps. On each step one of the subset $\Pi_k, k = 1,\ldots,|B|$ is chosen randomly. And the local operator $\Theta(x)$ is applied synchronously to all cells of chosen partition.

The choice order of cells belonging to the same partition X_k is unimportant, because condition (12) ensures that application $\Theta(x)$ to cells from different blocks B are independent.

Although, correctness of block-synchronous transformation and coincidence of ACA and BSCA simulation results are shown in some papers [9, 15], the equivalence of ACA and its block-synchronous transformation is not proved in general case. Moreover, it is shown that there are CA-models, in which the transformation of asynchronous mode to the block-synchronous leads to changes of its evolution. Therefore, it is necessary to verify the plausibility of block-synchronous transformation for pattern formation TCA.

4.2 Applicability of Block-Synchronous Transformation for Totalistic CA Model with Second Layer

Verification of block-synchronous transformation applicability for $\aleph_S$ parallel implementation is performed by comparison of ACA and BCSA evolutions. The most important application of TCA is porous materials synthesis, therefore, the following porous medium characteristics are further used as comparison criteria.

1. *Percolation* in the horizontal directions ($Per(0)$), the numbers of connected components, containing two cells with states equal to "zero", belonging to opposite (the left and right) borders of the cellular array.
2. The *numbers of connected components* formed by cells with states "one" ($L(1)$) and "zero" ($L(0)$).
3. *Porosity*, the ratio of the number of cells with state "zero" to the cellular array size: $Por = N(0)/|X|$.

These characteristics were computed for a number ($n = 100$) of computer experiments for the same CA but different initial values of random number generator. For the sample of obtained characteristics values the following statistics are calculated: mathematical expectations (M_α, M_β), dispersion (D_α, D_β), confidence intervals. Moreover, the root-mean-square difference between the characteristics obtained for ACA and BSCA is computed by formula (14):

$$E = \sqrt{\frac{\sum_{i=1}^{V}(\xi_{ACA}^i - \xi_{BSCA}^i)^2}{V}} \tag{14}$$

where ξ^i_{ACA} and ξ^i_{BSCA} are values of characteristics obtained by i-th experiment, V is the amount of sampling being equal to the number of experiments n. And also, hypotheses about the uniformity of these statistics distribution (H_0) was checked for each characteristic by χ^2-test of homogeneity [16]. For checking of H_0, grouping of characteristic' values ξ^i_{ACA} and ξ^i_{BSCA} is performed, the number of groups is equal to 10. Then, the number of ξ^i_{ACA} and ξ^i_{BSCA} belonging to each group is computed and χ^2-test is applied with a significance level of 0.001.

Experiments are performed for asynchronous and block-synchronous $\aleph_S$ with $|X^3| = 105 \times 105 \times 105$ cells and $B = T_7$. $\aleph_{TCA}$ uses the following values of weight coefficients $n = -0.5$, $n_1 = -0.1$, $p = 2$ and initial state $\Omega(0)$ being irregular distribution of cells with state "one" (Ω_3). Ω_3 is formed as follows: the cellular array is divided by three parts along the horizontal axis. In each part cells with state "one" are allocated with the probabilities $P^1_{\Omega_3} = 0.01$, $P^2_{\Omega_3} = 0.1$, $P^3_{\Omega_3} = 0.3$, respectively. Example of irregular distribution is presented in Fig. 5. Local operator of $\aleph_{TCA}$ calculates new values of cells states using function $f_2(u_k)$ (9). CA of second layer $\aleph_{2L}$ evolves from $\Omega(0)$, having all cells with state "one", and with the following values of probabilities of self-ignition and growth of tree: $P_f = 0.5$, $P_t = 0.00005$.

Table 1 presents statistics obtained for described above $\aleph_S$. For all characteristics except $Per(0)$ hypotheses about the uniformity of statistics distribution is rejected. Also, difference between dispersions of these characteristics are sizeable. These results indicate that evolution of block-synchronous $\aleph_S$ differs from that of asynchronous $\aleph_S$.

Table 1. The statistics of characteristics obtained by evolution of $\aleph_S$ with asynchronous and block-synchronous mode for $B = T_7$

	$\frac{M_\alpha - M_\beta}{M_\alpha}$	$\frac{D_\alpha - D_\beta}{D_\alpha}$	$\frac{E}{M_\alpha}$	H_0
$Per(0)$	0.0004	0.9792	0.0231	+
$L(0)$	0.0484	35.7183	0.5424	-
$L(1)$	0.0278	18.1661	0.0945	-
Por	0.0011	2.3086	0.0051	-

For all that, computer experiments reveal that evolution of asynchronous CA becomes more close to one of block-synchronous CA when block size $|B|$ is increased. Table 2 presents statistics obtained for the same $\aleph_S$ as in Table 1 with the exception of $|B|$. When $|B| = |B_{15}| = 15 \times 15 \times 15$ cells, values of the mathematical expectations for characteristics obtained by ACA and BSCA, except $L(0)$, differ by less than 10^{-3}. With further increase of block size $|B| = |B_{21}| = 21 \times 21 \times 21$ differences between the values of all statistics become insignificant and H_0 for all characteristics is accepted.

Table 2. The statistics of characteristics obtained by evolutions of $\aleph_S$ with asynchronous and block-synchronous mode for B_{15} and B_{21}

$\|B\|$	$15 \times 15 \times 15$				$21 \times 21 \times 21$			
	$\frac{M_\alpha - M_\beta}{M_\alpha}$	$\frac{D_\alpha - D_\beta}{D_\alpha}$	$\frac{E}{M_\alpha}$	H_0	$\frac{M_\alpha - M_\beta}{M_\alpha}$	$\frac{D_\alpha - D_\beta}{D_\alpha}$	$\frac{E}{M_\alpha}$	H_0
$Per(0)$	0.0042	0.1645	0.0181	+	0.0015	0.0260	0.0183	+
$L(0)$	0.0036	2.8083	0.2048	-	0.0026	0.9978	0.1529	+
$L(1)$	0.0002	0.9244	0.0335	+	0.0001	0.8363	0.0312	+
Por	0.0003	0.5120	0.0035	+	0.0002	0.1195	0.0032	+

As follows from above, asynchronous and block-synchronous modes of totalistic CA $\aleph_S$ lead to different evolutions when block size $|B| = |T_K|$, but when $|B|$ is increased evolutions of ACA and BSCA coincide. Thus, transformation of asynchronous mode to block-synchronous one for totalistic CA is plausible on the assumption of selection of necessary block size.

4.3 Parallel Implementation of Block-Synchronous TCA with Second Layer

Parallel implementation of block-synchronous TCA-model with second layer is performed using the domain decomposition. The cellular arrays are divided into n domains according to the number of available processors. Each processor computes new states of its domain's cells simultaneously with others processors. Moreover, two threads are created on each processor for simultaneous calculation of new states of the cells of $\aleph_{TCA}$ and $\aleph_{2L}$. According to the algorithm of block-synchronous transformation data exchange is performed after each stage. Thus, parallel algorithm $\aleph_S$ as follows:

1. Ω_{TCA} and Ω_{2L} are divided into n domains Ω_{TCA_r} and Ω_{2L_r}, $r = 1, \ldots, n$ which are distributed between n processors;
2. on each stage k, $k = 1, \ldots, |B|$:
   (a) one of the subsets Π_k is chosen randomly for all processors;
   (b) each r-th processor ($r = 1, \ldots, n$):
      i. copies Ω_{2L_r} to Ω'_{2L_r};
      ii. creates two parallel threads calculating states of cells $\mathbf{y} \in \Omega_{2L\_r}$ according to (11) and $\mathbf{x} \in \Omega_{TCA\_r}$ according to (7) with $v_k \in \Omega'_{2L\_r}$, simultaneously;
      iii. exchanges with boundary cells' values of both arrays Ω_{TCA_r} and Ω_{2L_r}.

The computations for $\aleph_S$ with parameters described in Sect. 4.2 have been performed on cluster $MVS-100K$ of JSCC RAS, Moscow. Simulation on one processor requires a lot of time, therefore, a weak scaling test is performed. A weak scaling test fixes the amount of work ($|\Omega_r| = 343 \times 343 \times 343$) per processor and compares the execution time over number of processors. Table 3 shows the weak scaling efficiency $Q(n) = \frac{T_1}{T_n}$, where T_n is computation time on the n processors. Since each processor has the same $|\Omega_r|$, in the ideal case the execution time should remain constant and $Q(n) = 1$. In the experiments $Q(n)$ decreases to 0.78 for $n = 16$ and then almost does not change with increasing n.

Table 3. Parallel implementation characteristics of BSCA

n	1	16	64	128	256	512	1024
$Q(n)$	1	0.78	0.76	0.75	0.75	0.74	0.73

5 Conclusion

The parallel composition of totalistic CA and asynchronous CA ($\aleph_S$) intended for forming of stable patterns similar to the porous materials is proposed and investigated. It is shown that TCA with second layer for different model parameters: initial conditions and values of weighted coefficients, enables to synthesize various porous media morphologies. In addition, the porous medium characteristics such as percolation degree, porosity, the number of connected components are calculated in the course of simulation. On the basis of their values analysis, the desired porous medium morphology can be selected.

For synthesis of three-dimensional real porous materials parallel implementation of asynchronous $\aleph_S$ is performed by means of block-synchronous transformation. Comparative analysis of evolutions of $\aleph_S$ asynchronous and block-synchronous modes has proved the plausibility of block-synchronous transformation for pattern formation TCA with appropriate block size. The efficiency of the parallel implementation of $\aleph_S$ with $|\Omega_r| = 343 \times 343 \times 343$ cells on the $n = 1024$ processors is greater than 0.7.

References

[1] Turing, A.M.: The Chemical Basis of Morphogenesis. Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences 237(641), 37–72 (1952)

[2] Zhabotinskiy, A.M., Zaikin, A.N.: Concentration wave propogation in a two-dimensional, liquid phase self-oscillating system. Nature 225, 535 (1970) (in Russian)

[3] Young, D.A.: A local activator-inhibitor model of vertebrate skin patterns. Theory and applications of cellular automata. Advanced series on complex systems, vol. 1, pp. 320–327. World Scientific Publishing Co. Pte. Ltd. (1986)

[4] Vanag, V.K.: Dissociative structures in reaction-diffusion systems, p. 300. Institute of computer researches, Regular and chaotic dynamics, Izhevsk (2008) (in Russian)

[5] Wolfram, S.: A New Kind of Science. Wolfram Media (2002)

[6] Wolfram, S.: Cellular Automata as Simple Self-Organizing Systems. Caltech preprint CALT-68-938 (1982), http://www.stephenwolfram.com/publications/articles/ca/82-cellular/index.html
[7] Chua, L.O.: CNN: a paradigm for complexity. Series on nonlinear science. Series A, vol. 31, p. 320. World Scientific Publishing Co. Pte. Ltd., Singapore (1998)
[8] Bandman, O.: Using Cellular Automata for porous media simulation. The Journal of Supercomputing 57(2), 121–131 (2011)
[9] Bandman, O.: 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)
[10] 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)
[11] Kalgin, K.V.: Similarity of the evolutions of cellular automaton in asynchronous and block-synchronous modes. In: New Information Technologies in the Study of Complex Structures: Proceedings of the 9-th Russian Conference, p. 21. NTL, Tomsk (2012) (in Russian)
[12] Achasova, S., Bandman, O., Markova, V., Piskunov, S.: Parallel Substitution Algorithm. Theory and Application. World Scientific Publishing Co. Pte. Ltd., Singapore (1994)
[13] Sharifulina, A.: Investigation of Stable Patterns Formed by Totalistic Cellular Automata Evolution. In: Sirakoulis, G.C., Bandini, S. (eds.) ACRI 2012. LNCS, vol. 7495, pp. 161–170. Springer, Heidelberg (2012)
[14] Bandman, O.: Using multi core computers for implementing cellular automata systems. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 140–151. Springer, Heidelberg (2011)
[15] Sharifulina, A., Elokhin, V.: Simulation of Heterogeneous Catalytic Reaction by Asynchronous Cellular Automata on Multicomputer. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 204–209. Springer, Heidelberg (2011)
[16] Ivchenko, G.I., Medvedev, Y.I.: Introduction to mathematical statistics. Moscow - LKI. P. 600 (2010) (in Russian)

cupSODA: A CUDA-Powered Simulator of Mass-Action Kinetics*

Marco S. Nobile[1], Daniela Besozzi[2,5], Paolo Cazzaniga[3,5], Giancarlo Mauri[1], and Dario Pescini[4]

[1] Università degli Studi di Milano-Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Viale Sarca 336, 20126 Milano, Italy
{nobile,mauri}@disco.unimib.it

[2] Università degli Studi di Milano
Dipartimento di Informatica
Via Comelico 39, 20135 Milano, Italy
besozzi@di.unimi.it

[3] Università degli Studi di Bergamo
Dipartimento di Scienze Umane e Sociali
Piazzale S. Agostino 2, 24129 Bergamo, Italy
paolo.cazzaniga@unibg.it

[4] Università degli Studi di Milano-Bicocca
Dipartimento di Statistica e Metodi Quantitativi
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy
dario.pescini@unimib.it

[5] Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti"
Consiglio Nazionale delle Ricerche
Viale Manzoni 30, 00185 Roma, Italy

Abstract. The computational investigation of a biological system often requires the execution of a large number of simulations to analyze its dynamics, and to derive useful knowledge on its behavior under physiological and perturbed conditions. This analysis usually turns out into very high computational costs when simulations are run on central processing units (CPUs), therefore demanding a shift to the use of high-performance processors. In this work we present a simulator of biological systems, called cupSODA, which exploits the higher memory bandwidth and computational capability of graphics processing units (GPUs). This software allows to execute parallel simulations of the dynamics of biological systems, by first deriving a set of ordinary differential equations from reaction-based mechanistic models defined according to the mass-action kinetics, and then exploiting the numerical integration algorithm LSODA. We show that cupSODA can achieve a 112× speedup on GPUs with respect to equivalent executions of LSODA on CPUs.

* This work is partially supported by Regione Lombardia, project "Network Enabled Drug Design (NEDD)", and by the Research Infrastructure "SYSBIO Centre of Systems Biology".

1 Introduction

In the last decades, the use of mathematical modeling and computational methods have fostered our comprehension of the functioning of biological systems, thanks to the advantages that these techniques present over conventional experimental biology in terms of cost, ease to use and rapidity. Given a mathematical model describing the physical or logical interactions between the components of a biological system, computer algorithms can be exploited to test and analyze the model, and to make predictions on the way the system behaves in normal or perturbed conditions. In this context, simulation algorithms represent an essential tool to investigate the dynamics of biological systems: starting from distinct parameterizations of the model, different emergent behaviors can be reached; the intensive exploration of high-dimensional parameter spaces allows to understand the system functioning across a wide spectrum of natural conditions, as well as to derive statistically meaningful properties. These issues play a fundamental role in standard computational investigations of biological systems [1, 2], which usually rely on methods such as parameter sweep analysis [3], sensitivity analysis [4], structure and parameter identifiability [5], parameter estimation [6–8] and reverse engineering of model topologies [9–12].

To serve these purposes, the dynamics of biological systems can be reproduced by means of deterministic or stochastic procedures, based either on numerical integration (e.g. Euler's or Runge-Kutta methods [13]) or on Markov processes (e.g. Gillespie's algorithm [14]), respectively. In the case of deterministic models, one assumes the availability of a system of ordinary differential equations (ODEs) – corresponding to the biochemical reaction rate equations – which overall describe how the concentration of each chemical species occurring in the system varies in time. To date, one of the most efficient algorithms to integrate a set of ODEs is LSODA, a solver able to automatically recognize stiff[1] and non-stiff systems, and to dynamically select between the most appropriate integration procedure: Adams' method in the absence of stiffness, and the Backward Differentiation Formulae otherwise [15].

In this work we present a simulator for biological systems which relies on an efficient implementation of LSODA on Graphics Processing Units (GPUs), to the aim of efficiently execute a large number of parallel deterministic simulations at a considerable reduced computational cost with respect to Central Processing Units (CPUs). The rationale behind the use of GPUs in the context of scientific computing – where CPUs have traditionally been the standard workhorses – is that, when several batches of simulations need to be executed, the necessary computing power usually overtakes the capabilities of standard desktop computers, therefore requiring high-performance computing solutions. Indeed, after the introduction of general-purpose GPUs and CUDA (Nvidia's GPU programming language), the adoption of these graphics engines have largely increased,

[1] A system of ODEs is said to be stiff if it is characterized by two well-separated dynamical modes – determined by fast and by slow reactions – the fastest of which is stable [14].

especially in scientific applications related to Bioinformatics, Systems Biology and Computational Biology (see an overview in [16–18]).

Despite the remarkable advantages in terms of computational speedup, computing with GPUs requires the development and the implementation of *ad hoc* algorithms, since GPU-based programming substantially differs from CPU-based computing. As a consequence, scientific applications on GPUs might undergo the risk of remaining a niche for few specialists. To avoid such limitations, several packages and software tools have recently been released in the aforementioned research fields (see, e.g., [18–20]), so that also users with no knowledge of GPUs hardware and programming can have the chance to access the computing power of graphics engines.

cupSODA, the simulator for mass-action kinetics models that we present in this work, fulfills these requirements. It allows not only to easily execute efficient simulations of deterministic models running a GPU-based version of the LSODA algorithm, but also to circumvent the need of manually defining ODEs to describe the biological system under investigation. More precisely, cupSODA is able to automatically derive a system of ODEs – and then to perform their numerical integration – starting from a set of biochemical reactions, which describe the molecular interactions between all the species in the system. This way, by just providing mechanistic reaction-based models of biological systems, together with a proper parameterization, any user – having or not either GPU programming skills or mathematical modeling expertise – will be able to run parallel simulations of mass-action kinetics systems at reduced costs.

In what follows, we introduce the formal representation of mass-action kinetics, describe the implementation of cupSODA, and finally present the computational speedup obtained for the parallel simulations of three biological systems. We conclude the paper with some considerations and hints for future research directions.

2 Mass-Action Kinetics and Deterministic Simulation

The fundamental empirical law that governs biochemical reaction rates is called the *law of mass action*: it states that, in a dilute solution, the rate of an elementary reaction (i.e., a reaction with a single mechanistic step) is proportional to the product of the concentration of its reactants raised to the power of the corresponding stoichiometric coefficient. Given a set of biochemical reactions (each one characterized by its own kinetic constant k), it is possible to determine the corresponding set of rate equations – one for each chemical species involved in the set of reactions – by considering all the modifications of each species defined by means of all reactions. For instance, given the following set of biochemical reactions

$$S_1 + S_2 \underset{k_2}{\overset{k_1}{\rightleftarrows}} S_3 \overset{k_3}{\rightarrow} S_4,$$

and denoting by $[S_i]$ the concentration of species S_i, the rate equation of species S_3 is

$$\frac{d[S_3]}{dt} = \frac{d[S_3^+]}{dt} - \frac{d[S_3^-]}{dt} = k_1[S_1][S_2] - (k_2 + k_3)[S_3],$$

which is given by the sum of the rates of production and degradation of S_3 (denoted by S_3^+ and S_3^-, respectively), each one determined by multiplying the concentrations (raised to the power of 1, in this case) of all the species involved in these reactions together with the kinetic constants of such reactions.

So doing, by using the law of mass action, we can derive the expression for the rate of change of all the chemical species occurring in the system. Therefore, any biochemical system defined by means of a mechanistic reaction-based model can be also formalized as a set of coupled (non-linear) first order ordinary differential equations (ODEs). In general, the analytical solution of these systems of ODEs is hard to find (apart from the most simple cases); however, it is possible to determine their dynamics by exploiting numerical integration algorithms, such as Euler's or Runge-Kutta methods [13]. These methods require as input the set of ODEs, along with the set of kinetic constants and the initial concentrations of the chemical species.

One of the most efficient numerical integration algorithms – that can also be used for stiff systems – is called LSODA [15]; however, when a large number of simulations is needed to carry out the analysis of a biological system, LSODA turns out to be very time consuming if these simulations are run in a sequential manner on the CPU. Therefore, a novel implementation of LSODA that exploits a parallel architecture as the GPU would be extremely advantageous.

3 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model introduced by Nvidia in 2006, to give the programmers a way to exploit GPUs in general-purpose computational tasks (GPGPU computing). The GPGPU computing is a low-cost and energy-wise alternative to the traditional high-performance computing infrastructures, which gives access to tera-scale computing on a common workstation. Because of the innovative architecture and the intrinsic limitations of the GPUs, the challenge of GPGPU computing is that the direct porting of an application may be unfeasible, or may not fully exploit their computational power and massive parallelism [21].

CUDA's architecture combines the Single Instruction Multiple Data (SIMD) architecture with a flexible multi-threading by automatically taking care of any conditional divergence between threads. Using CUDA's naming conventions, the programmer implements the *kernel*, that is, a C/C++ function[2], which is loaded from the host (the CPU) to the devices (one or more GPUs), and replicated in many copies named *threads*. Threads can be organized in three-dimensional

[2] Even though templates, overloading and basic classes can be exploited, CUDA's support for C++ is still partial.

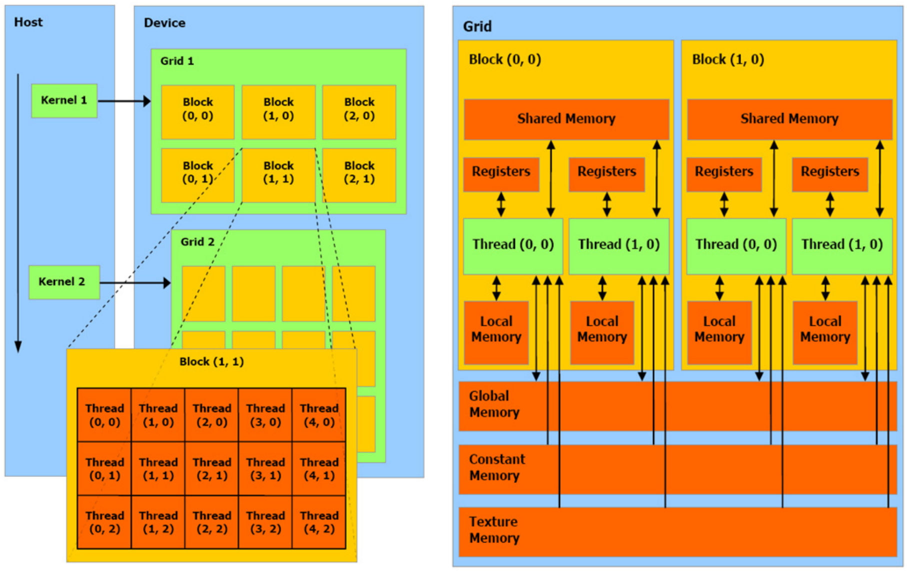


Fig. 1. Schematic description of CUDA's architecture, in terms of threads and memory hierarchy. *Left side.* Thread organization: a single kernel is launched from the host (the CPU) and is executed in multiple threads on the device (the GPU). Threads can be organized in three-dimensional structures named blocks which can be, in turn, organized in three-dimensional grids. The dimensions of blocks and grids are explicitly defined by the programmer. *Right side.* Memory hierarchy: threads can access data from many different memories with different scopes. Registers and local memories are private for each thread. Shared memory let threads belonging to the same block communicate, and has low access latency. All threads can access the global memory, which suffers high latencies, but it is cached since the introduction of the Fermi architecture. Texture and constant memory can be read from any thread and are equipped with a cache as well; in this work we exploit the constant memory. Figures are taken from Nvidia's CUDA programming guide [22].

structures named *blocks* which, in turn, are contained in three-dimensional *grids* (see a schematic description in Figure 1, on the left). Whenever the host runs a kernel, the GPU creates the corresponding grid and automatically schedules each block on one free streaming multi-processor available on the GPU, a solution that allows a transparent scaling of performances on different devices. Moreover, if more than one GPU is present on the machine, the workload can be also distributed by launching the kernel on each GPU.

Threads can access data from different kinds of memory. In the present work, we exploit the *global memory* (accessible from all threads), the *shared memory* (accessible from threads belonging to the same block), the *local memory* (registers and arrays, accessible from owner thread), and the *constant memory* (cached and not modifiable). A schematic representation of CUDA's memory hierarchy is shown in Figure 1, on the right. The global memory is usually very large (thousands of MBs) but suffers of high access latencies, even though – concerning the Fermi architecture – it is equipped of L1 cache that mitigates this problem. Viceversa, the shared memory is faster but much smaller (i.e., 49152 bytes for each multi-processor, since the introduction of the Fermi architecture). In order to obtain the best performances, the shared memory should be exploited as much as possible but, being a very limited resource on each multi-processor, it poses a limitation on the blocks' size.

4 Implementation of cupSODA

The development of cupSODA was motivated by the need of systematic analyses of biological systems consisting in the execution of large batches of simulations. Since in standard analysis, based on an intensive search within the parameters space, all simulations are mutually independent, we decided to implement on a parallel architecture one of the most efficient numerical integration algorithms for ODEs available to date. In particular, cupSODA relies on a C version of LSODA [15], ported and adapted to the CUDA architecture in order to be run on the GPU; CUDA's massive parallelism is exploited to execute different and independent simulations in each thread.

LSODA was designed to solve differential systems in the canonical form (i.e., defined as a set of equations of the form $\frac{dX}{dt} = f(X, t)$, where here X represents the concentration of a chemical species), whereby the developer is supposed to specify the system of ODEs by implementing a custom function that is passed to the algorithm. Moreover, in order to speed up the computation when dealing with stiff systems, the Jacobian matrix associated to the system must be implemented as a custom function as well.

cupSODA, on the other hand, is designed to the purpose of being a *black-box* simulator, that can be easily used without any programming skills. In particular, cupSODA is constituted by a tool that automatically converts the mechanistic reaction-based model of a biological system into the corresponding set of ODEs, according to the mass-action kinetics [23], and then it automatically encodes the ODEs system and its relative Jacobian matrix as arrays. The two arrays are loaded into the GPU, automatically parsed and implemented as custom functions. In order to encode each term of each ODE into a linear data structure, without any loss of information, the array contains the following data: the sign of the term of the equation; the index of the kinetic constant associated to the term; the number of the chemical species involved in the term and their corresponding indeces (see an example in Figure 2).

The terms of all ODEs are linked together in these arrays. To efficiently parse the arrays inside the GPU, we use two additional arrays storing the offsets of each equation, so that the parsing algorithm consists in the function given in Figure 3. A similar function is defined for the parsing of the Jacobian matrix.

The cupSODA simulator was designed to speed up the time-consuming computational tasks typical of Systems Biology [1, 2], which rely on the repetition of large numbers of simulations in perturbed conditions, generally realized by

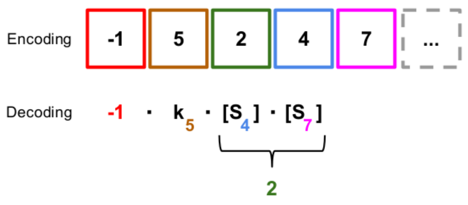


Fig. 2. Example of our encoding methodology, showing the internal representation of a single term of an ODE, used for the parsing device-side. From the left to the right, the values of the array have the following semantics: the sign of the term (red); the index of the kinetic constant associated to the term (brown); the number of species involved in this term (green) and the indexes of these species (blue and violet). All the terms of all the ODEs are linked together in a single one-dimensional array.

varying the initial concentrations of chemical species or the value of the kinetic constants. To fulfill these necessities, cupSODA is able to launch multiple threads which run independent parallel simulations based on the same model, with each thread exploiting its own parameterization and initial conditions. To this aim, parameters and initial conditions are contained in coalesced arrays, a strategy that allows a faster fetching of data from the global memory [24].

An additional feature of cupSODA is that it allows to easily compare the outcome of simulations with any available experimental data. To be more precise, let us assume that a set of laboratory measurements (e.g., the concentration of some chemical species) has been obtained with a certain sampling, denoted by $\mathbf{t} = \{t_0, \ldots, t_F\}$, and that the user wishes to determine the corresponding simulated concentration of the same species at these time points. To this aim, cupSODA invokes the LSODA kernel $F - 1$ times, at each time the kernel is run over a time window of length $\Delta t = t_i - t_{i-1}$, $i = 1, \ldots, F$, and the simulated concentration values of output species are stored at the end of each Δt.

Being the data transfer between host and device very time consuming, all the temporary results are allocated on the GPU, and the data are then read back as soon as the simulation is completed. To obtain a further reduction of the memory latencies, the current state and time of each simulation are stored into the shared memory, while all the constants values (e.g., number of reactions and chemical species in the model, length of ODEs and Jacobian arrays, etc.) and LSODA settings are stored into the constant memory.

Since the amount of shared memory is limited on each streaming multiprocessor, cupSODA automatically calculates the number of threads per block and blocks per grid. We allocate the states of the system and the current time as double precision floating point values, so that the consumption of shared memory of each thread is $M = 8 \times (N + 1)$ bytes of shared memory during the ODE integration, where N is the number of chemical species in the system. We automatically determine the threads-per-block value as $T_{pb} = \lfloor \frac{SM}{M} \rfloor$, where SM is the shared memory available on the GPU, so that the number of resulting blocks is $B = \left\lfloor \frac{T_{tot}}{T_{pb}} \right\rfloor$, where T_{tot} is the number of total threads requested by the user.

```
function decode_ODEs ( encoded_ODE, ODE_offset ):
  dX = [0, ..., 0]
  position = 0
  for i = 0 to ODE_offset.length do:
    ode_value = 0
    while( position<ODE_offset[i] ) do:
      term = encoded_ODE[position]
      term *= k[encoded_ODE[position+1]]
      species = encoded_ODE[position+2]
      for s = 0 to species do:
        term *= X[encoded_ODE[position+3+s]]
      end for
      position += (3+species)
      ode_value += term
    end while
    dX[i] = ode_value
  end for
  return dX
end function
```

Fig. 3. Pseudocode of the parsing algorithm, exploited device-side by cupSODA for the decoding of the ODEs. A similar code is used for the decoding of the Jacobian matrix.

LSODA requires additional parameters for its functioning, the most relevant being the absolute and relative error tolerance values (denoted by AET and RET, respectively). These values can be either scalar values or specific vectors for each ODE: cupSODA accepts all the combinations. Moreover, these values can be specified for each individual thread, allowing the user to simulate the same system with different tolerances and compare their outcomes.

5 Results

In this section we compare the performances of cupSODA against a reference sequential simulator, the COmplex PAthways SImulator (COPASI [25]), in order to show the suitability of our method when the execution of a large number of simulations is necessary. To this purpose, we performed several tests, consisting in running different batches of independent simulations of three biological models characterized by an increasing complexity:

1. the Michaelis-Menten (MM) enzymatic kinetics, consisting in 3 reactions and 4 chemical species [26]. For this model, both cupSODA and COPASI were set as follows: RET= 10^{-10}, AET= 10^{-10};
2. a model of gene expression in prokaryotic organisms (PGN), consisting in 8 reactions and 5 chemical species [27]. For this model, both cupSODA and COPASI were set as follows: RET=10^{-10}, AET=10^{-10};

3. the Ras/cAMP/PKA signaling pathway in the yeast *S. cerevisiae*, consisting in 30 reactions and 28 chemical species [3, 28]. For this model, both cupSODA and COPASI were set as follows: RET=10^{-10}, AET=10^{-14}.

For every test, the maximum number of internal steps allowed during each call of LSODA was set to 10000. In addition, during each test we stored a dynamics consisting of 100 time instants – uniformly sampled over the whole simulation time – keeping track of the overall running time.

The GPU used for the tests is a Nvidia GeForce GTX 590, a video card with Fermi architecture equipped with 2×16 streaming multiprocessors for a total of 1024 cores and a theoretical limit of 2.48 Tflops. The performances of the GTX 590 were compared with a quad-core CPU Intel Core i7-2600 with a clock rate of 3.4 GHz and capable of 83.6 Gflops. Because of their architectural differences (e.g., the multiple cache levels of CPUs), GPUs and CPUs are difficult to be compared. Moreover, the theoretical peak limits of GPUs can be achieved only by implementing kernels that maximize parallelism and GPU occupancy: this is achieved by completely exploiting the SIMD computational model, that is, by avoiding conditional branches in the code, which cause the serialization of the execution. In addition, the occupancy of the GPU is affected by the usage of registers and shared memory, which are both limited resources on each streaming multiprocessors. For these reasons, the theoretical computational power peak of GPUs is often hard to be reached; nonetheless, we compare here the performances of these two hardware devices since they are well representative of hardware components typically found in personal computers.

The tests were performed on a system running the operating system Microsoft Windows 7, CUDA version 4.2, COPASI 4.8 (build 35).

Figure 4 reports a direct comparison of the running times of LSODA implemented in COPASI and cupSODA, obtained by performing an increasing number of simulations for the three biological models. These results show that our parallelized implementation of LSODA largely outperforms the serial counterpart implemented in COPASI. In particular, in the case of 10^5 parallel threads for the MM model (Figure 4, top graphic), the computational cost on the GPU is nearly two orders of magnitude smaller than the CPU: 3.358 seconds *vs.* 289.335 seconds, which corresponds to a $86\times$ speedup.

In the case of the PGN model (Figure 4, middle graphic), the execution of 10^5 simulations takes 43.821 seconds on the CPU, while it takes just 0.391 seconds on the GPU, resulting in a $112\times$ speedup. Interestingly, the running time of 10 simulations of the PGN model is quite identical on the two architectures: 0.047 seconds on the GPU *vs.* 0.046 seconds on the GPU.

Finally, concerning the Ras/cAMP/PKA model (Figure 4, bottom graphic), the execution of 10^5 simulations lasts 6133.3 seconds on the CPU, while it takes 268.58 seconds on the GPU, resulting in a $23\times$ speedup. Here, the results clearly indicate that cupSODA turns out to be convenient only when more than 10 parallel simulations are required, since the computational cost to perform a small number of simulations is lower on the CPU.

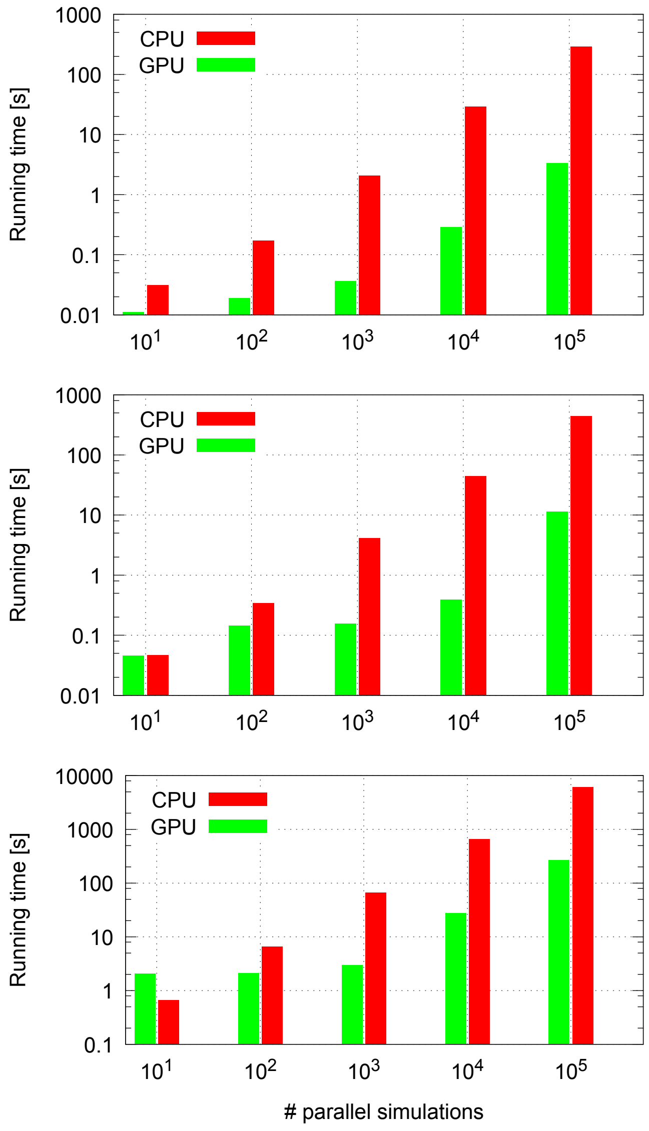


Fig. 4. Comparison between the computational time of cupSODA (green histograms) and COPASI (red histograms) for simulating the MM model (top graphic), the PGN model (middle graphic), and the Ras/cAMP/PKA model (bottom graphic). The y-axis are in logarithmic scale. The results clearly show that cupSODA largely outperforms the serial implementation of LSODA available in COPASI, except in the case of the execution of a few simulations for the Ras/cAMP/PKA model.

As a last test, we investigated the impact of the different memories on cupSODA, by executing 10^5 simulations of the Ras/cAMP/PKA pathway exploiting the shared memory and the global memory. The simulations took, respectively, 27.501 seconds and 54.93 seconds, showing the benefit deriving from the use of low latency memories; in both cases, the result is far better than the 651.741 seconds of COPASI but, by exploiting the shared memory, cupSODA is twice as fast as the naive porting of LSODA.

6 Conclusions and Future Work

The exploitation of computational methods for *in silico* analysis of biological systems has heightened the need for novel and efficient algorithms, in order to carry out fast simulations and to analyze the emergent behavior of these complex systems.

GPUs represent a suitable technology for this kind of problems, thanks to their high-performance parallel computing capabilities combined with very low costs. Presently, the bottleneck of such a large potentiality resides in the programming skills required to implement GPU-based algorithmic methods, and to handle specific features of GPU computing, as the efficient usage of memory or the communication bandwidth between GPU and CPU. As a matter of fact, to fully exploit the underlying SIMD architecture and the memory hierarchy, the algorithms must be heavily restructured or purposely designed. Moreover, a direct porting from the CPU source-code to CUDA is most of the times unfeasible, because of the different architectures and the limited programming capabilities allowed by GPU kernels.

In this work we presented cupSODA, a GPU-powered simulator of biochemical system based on mass-action kinetics. cupSODA was designed to offer a black-box solution, capable of automatically translating the reaction-based model of a biological system into a set of ODEs. cupSODA relies on a numerical integrator for ODEs, called LSODA, which we implemented as a CUDA kernel starting from a C implementation of LSODA, in order to exploit the massive parallel capabilities of modern GPUs, thus achieving a relevant reduction of the computational time usually required to execute a huge number of independent simulations. The mutual independence of the simulations allows to fully exploit the underlying SIMD architecture; moreover, cupSODA benefits from an additional speedup, thanks to our choice of storing each system state into the low-latency shared memory.

A previous GPU implementation of LSODA algorithm was proposed in the cuda-sim library [19], a package for the Python language that provides GPU-accelerated biochemical simulations. The aim and design of cuda-sim are very different from cupSODA, as the latter does not require any code to be written by the user to run the simulations. Moreover, cuda-sim relies on a *just-in-time* technique, whereby the code for LSODA that will be executed on the GPU is automatically created and compiled at run-time. This is indeed a flexible and elegant solution, but adds a relatively long compilation time and requires the availability of a CUDA compiler, along with the CUDA toolkit, on the running machine. During the development of cupSODA, instead, we opted for the encoding of ODEs and the Jacobian matrix into linear arrays which are parsed device-side, without the need for an intermediate recourse to the CUDA driver API or any meta-programming techniques. Thanks to this design choice, cupSODA allows the immediate simulation of any biological system modeled according the mass-action kinetics, without the need for a complete re-compilation of the code. This is particularly appealing when the topology of the model is not static but needs to be continuously changed, for instance when it undergoes a reverse engineering process [9–11]: in such a case, cupSODA only needs to update the GPU representation of the ODEs and the Jacobian matrix, in order to be ready to run a massive number of simulations of the new model.

The results of the tests reported in this work, performed on three biological models of increasing complexity (Michaelis-Menten kinetics, gene expression in prokaryotic organisms, and the Ras/cAMP/PKA signaling pathway in yeast), show that cupSODA allows a relevant boost with respect to a reference CPU implementation. For instance, in the case of 10^5 simulations of the PGN model, we achieved a noticeable 112$\times$ speedup. Interestingly, the performances of cupSODA are identical to COPASI when 10 simulations of the PGN model are run, and are even worse in the case of 10 simulations of the Ras/cAMP/PKA model. These results indicate that, for biological systems consisting in many reactions and many species, our GPU implementation becomes more profitable than the CPU counterpart only if many parallel simulations are run, with a break-even that depends on the complexity of the system under investigation. Indeed, when performing demanding computational analysis such as, e.g., parameter sweep, parameter estimation or sensitivity analysis, the outstanding advantage of novel softwares as cupSODA clearly comes to light.

Our tests also showed that, in order to fully exploit the CUDA architecture, the memory hierarchy must be exploited as much as possible: as a matter of fact, by moving the state of the system from the global memory to the shared memory, the running time of cupSODA was halved. As an extension of this solution, we are currently working on a compact representations of the static data into the constant memory, in order to further increase the speedup of cupSODA.

The cupSODA software is available from the authors upon request.

References

1. Aldridge, B., Burke, J., Lauffenburger, D., Sorger, P.: Physicochemical modelling of cell signalling pathways. Nature Cell Biology 8, 1195–1203 (2006)
2. Chou, I., Voit, E.: Recent developments in parameter estimation and structure identification of biochemical and genomic systems. Mathematical Biosciences 219(2), 57–83 (2009)
3. Besozzi, D., Cazzaniga, P., Pescini, D., Mauri, G., Colombo, S., Martegani, E.: The role of feedback control mechanisms on the establishment of oscillatory regimes in the Ras/cAMP/PKA pathway in *S. cerevisiae*. EURASIP Journal on Bioinformatics and Systems Biology 2012(10) (2012)

4. Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S.: Global Sensitivity Analysis: The Primer. Wiley-Interscience (2008)
5. Raue, A., Kreutz, C., Maiwald, T., Bachmann, J., Schilling, M., Klingmüller, U., Timmer, J.: Structural and practical identifiability analysis of partially observed dynamical models by exploiting the profile likelihood. Bioinformatics 25(15), 1923–1929 (2009)
6. Dräger, A., Kronfeld, M., Ziller, M.J., Supper, J., Planatscher, H., Magnus, J.B., Oldiges, M., Kohlbacher, O., Zell, A.: Modeling metabolic networks in *C. glutamicum*: a comparison of rate laws in combination with various parameter optimization strategies. BMC Systems Biology 3(5) (2009)
7. Besozzi, D., Cazzaniga, P., Mauri, G., Pescini, D., Vanneschi, L.: A comparison of genetic algorithms and particle swarm optimization for parameter estimation in stochastic biochemical systems. In: Pizzuti, C., Ritchie, M.D., Giacobini, M. (eds.) EvoBIO 2009. LNCS, vol. 5483, pp. 116–127. Springer, Heidelberg (2009)
8. Nobile, M.S., Besozzi, D., Cazzaniga, P., Mauri, G., Pescini, D.: A GPU-based multi-swarm PSO method for parameter estimation in stochastic biological systems exploiting discrete-time target series. In: Giacobini, M., Vanneschi, L., Bush, W.S. (eds.) EvoBIO 2012. LNCS, vol. 7246, pp. 74–85. Springer, Heidelberg (2012)
9. Nobile, M.S., Cazzaniga, P., Besozzi, D., Pescini, D., Mauri, G.: Reverse engineering of kinetic reaction networks by means of cartesian genetic programming and particle swarm optimization. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2013) (In press, 2013)
10. Kentzoglanakis, K., Poole, M.: A swarm intelligence framework for reconstructing gene networks: Searching for biologically plausible architectures. IEEE/ACM Transactions on Computational Biology and Bioinformatics 9(2), 358–371 (2012)
11. Koza, J.R., Mydlowec, W., Lanza, G., Yu, J., Keane, M.A.: Automatic computational discovery of chemical reaction networks using genetic programming. In: Džeroski, S., Todorovski, L. (eds.) Computational Discovery 2007. LNCS (LNAI), vol. 4660, pp. 205–227. Springer, Heidelberg (2007)
12. Ando, S., Sakamoto, E., Iba, H.: Evolutionary modeling and inference of gene network. Information Sciences 145(3-4), 237–259 (2002)
13. Butcher, J.C.: Numerical Methods for Ordinary Differential Equations. John Wiley & Sons (2003)
14. Gillespie, D.T.: Stochastic simulation of chemical kinetics. Annual Review of Physical Chemistry 58, 35–55 (2007)
15. Petzold, L.: Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. SIAM Journal of Scientific and Statistical Computing 4(1), 136–148 (1983)
16. Demattè, L., Prandi, D.: GPU computing for systems biology. Briefings in Bioinformatics 11(3), 323–333 (2010)
17. Payne, J., Sinnott-Armstrong, N., Moore, J.: Exploiting graphics processing units for computational biology and bioinformatics. Interdisciplinary Sciences, Computational Life Sciences 2(3), 213–220 (2010)
18. Harvey, M.J., De Fabritiis, G.: A survey of computational molecular science using graphics processing units. Wiley Interdisciplinary Reviews: Computational Molecular Science 2(5), 734–742 (2012)
19. Zhou, Y., Liepe, J., Sheng, X., Stumpf, M.P.H., Barnes, C.: GPU accelerated biochemical network simulation. Bioinformatics 27(6), 874–876 (2011)
20. Vigelius, M., Lane, A., Meyer, B.: Accelerating reaction-diffusion simulations with general-purpose graphics processing units. Bioinformatics 27(2), 288–290 (2011)

21. Farber, R.: Topical perspective on massive threading and parallelism. Journal of Molecular Graphics and Modelling 30, 82–89 (2011)
22. Nvidia: CUDA C Programming Guide v5.0 (2012)
23. Wolkenhauer, O., Ullah, M., Kolch, W., Kwang-Hyun, C.: Modeling and simulation of intracellular dynamics: choosing an appropriate framework. IEEE Transactions on Nanobiosciences 3(3), 200–207 (2004)
24. Nvidia: CUDA C Best Practices Guide (2012)
25. Hoops, S., Sahle, S., Gauges, R., Lee, C., Pahle, J., Simus, N., Singhal, M., Xu, L., Mendes, P., Kummer, U.: COPASI - a COmplex PAthway SImulator. Bioinformatics 22(24), 3067–3074 (2006)
26. Nelson, D., Cox, M.: Lehninger Principles of Biochemistry. W. H. Freeman Company (2004)
27. Wang, Y., Christley, S., Mjolsness, E., Xie, X.: Parameter inference for discretely observed stochastic kinetic models using stochastic gradient descent. BMC Systems Biology 4(1) (2010)
28. Cazzaniga, P., Pescini, D., Besozzi, D., Mauri, G., Colombo, S., Martegani, E.: Modeling and stochastic simulation of the Ras/cAMP/PKA pathway in the yeast *Saccharomyces cerevisiae* evidences a key regulatory function for intracellular guanine nucleotides pools. Journal of Biotechnology 133(3), 377–385 (2008)

Reconstructing Images with Nature Inspired Algorithms

Franciszek Seredynski[1,2] and Jaroslaw Skaruz[2]

[1] Cardinal Stefan Wyszynski University in Warsaw,
Department of Mathematics and Natural Sciences
Woycickiego 1/3, 01-938 Warsaw, Poland
fseredynski@gmail.com
[2] Siedlce University of Natural Sciences and Humanities,
Institute of Computer Science
3 Maja 54, 08-110 Siedlce, Poland
jaroslaw.skaruz@uph.edu.pl

Abstract. An approach based on an application of Nature inspired algorithms to the problem of reconstructing human face images from ones with only partial information is presented in the paper. The two-dimensional cellular automata (CA) is used to represent images, while genetic algorithm (GA) is used to discover CA rules which will be able to reconstruct an original image from e.g. destroyed or modified images. Conducted experiments show that found in the learning process CA rules allow to reconstruct images with 70% damaged pixels. Moreover, the results indicate that the found rules are useful to reconstruct the other images not presented during evolutionary learning process.

Keywords: cellular automata, image reconstruction, genetic algorithms.

1 Introduction

CA were initially developed by Ulam and von Neumann in the 1940s. They were working on a framework to study and investigate features of dynamical systems. Since that time there has been a lot of research done towards applications of cellular automata in many fields.

CA have often been used to demonstrate the full spectrum of dynamical behaviour, e.g. hydrodynamics, thermodynamics, A-life, synchronization, simple image processing and control problems. CA have also been found useful for simulating and studying phenomena such as ordering, turbulence, chaos, symmetry-breaking, and have had wide application in modelling systems in areas such as physics, computer networks, biology, and sociology. Because of the quite obvious similarities between 2-dimensional cellular automata and images (pixels mapping to cells) uniform cellular automata have been used quite a lot in some areas of image processing. In [9] the author presented an approach to noise filtering and thinning of binary images and sequential floating forward search strategy to find a rule. Hernandez and Herrmann [6] presented CA for elementary image enhancement whose behaviour is determined by operators called Lyapunov functionals. Their results over real images compared with a common use image processing software allow them to propose CA as the first level of enhancement. Popovici et al. [8] applied CA implemented on a parallel machine to solve noise removal and border detection in images. Finally CA were also extensively applied in medicine e.g. to generate image representation for biological sequences [13].

The main bottleneck of CA is a difficulty of constructing CA rules producing a desired behaviour. In some applications of CA one can design an appropriate rule by hand, based on partial differential equations describing a given phenomenon. However, it is not always possible. In the 90-ties of the last century Mitchell and colleagues proposed to use genetic algorithms to discover CA rules able to perform one-dimensional density classification task [7] and the synchronization task [4]. The results produced by Mitchell et al. were interesting and started development of a concept of automating rule generation using artificial evolution. Breukelaar and Back applied GAs [3] to solve the density classification problem as well as AND and XOR problem in two dimensional CAs. Swiecicka et al. used [11] GAs to find CA rules able to solve multiprocessor scheduling problem. Bandini et al. proposed [1] to use several Machine Learning techniques such as GAs, Support Vector Machines and neural networks to find automatically CA rules able to generate patterns, which are similar in some generic sense to those generated by a given target rule.

In this paper we present results of experiments concerning evolution of CA rules to perform image reconstruction task, which is related to image processing. The rest of the paper is organized as follows. Section 2 describes CA. Section 3 explains pattern reconstruction task in the context of CAs. Section 4 presents details of the GA designed to find CA rules. Experimental results are reported in section 5. The last section contains conclusions.

2 Cellular Automata Overview

CA are discrete computational models, which compose of a regular grid of cells. Each of these cells are in one of a finite number of possible states and are updated in parallel according to a rule. The state of a cell is determined by the previous states of a cell and its neighbourhood, which is a number of adjacent cells. Two types of neighborhood are commonly used: the von Neumann neighborhood (the four cells orthogonally surrounding the central cell) and the Moore one (the eight cells around the central cell).

There are many possible kinds of CA. Traditionally, CA are implemented as uniform CA, which means that all cells have the same state transition function. There are also papers considering non-uniform CA with cells having different state transition functions. The rule governing CA can take various forms e.g. totalistic, probabilistic. Large space of possible rules was also inspiration to the application of GAs to find a rule in drastically smaller time periods. One of the advantages of CAs is that, although each cell can contain even only one simple rule, the combination of a grid of cells with their local interaction leads to more sophisticated global behaviour. Although each cell has only a view of a part of the system in the form of its neighbourhood, local information is propagated at each iteration, which make impact on the overall CA system.

3 Image Reconstruction Problem

Image reconstruction problem defines a recovery operation of an original form of an image from its distorted version. The malformation may be due to noise, blurred shapes or damage of some number of pixels. The image enhancement is a broad field and various methods are used to restore the true image.

In this paper we consider image reconstruction problem from different perspective. Suppose that one day, the police intercepted the offender in the act. According to the standard procedure he was taken to the police station and made him face images. Unfortunately, the suspected person managed to escape during transport to the court. Announced the search with just a photo taken earlier. Offender to avoid arrest changes its appearance with headgear and various kinds of glasses. Figure 1 presents example of a suspected person's face and his various aspects of face.

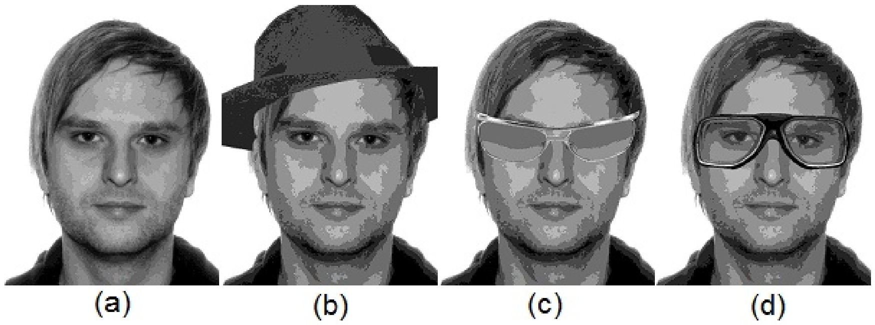


Fig. 1. An instance of the image reconstruction: original image (a), the images presenting the same face but with some modifications (b) - (d).

This article presents a new idea to solve the problem of image reconstruction. This idea consists in the use of cellular automata, whose task is to reconstruct the original picture of a human face based on the image with a certain number of defective pixels. Due to the large number of potential rules controlling CA, the task of searching good quality rules is realized by the GA.

4 Application of GA for Searching CA Rules

In this section we present how GA is used for searching rules for CA. Our assumption is that a given image is defined on a two-dimensional CA and one pixel corresponds to one cell of a CA. Common properties of images are the number of colors and the resolution. They are represented in CA in the form of possible states of cells and the size of CA, accordingly. In our experiments we used images with 300x400 pixels with eight colors. As a consequence our CA is 300x400 cells size and each cell can contain one of nine values representing a color of a given pixel. The number in the range between zero and seven reflects a color tone and value of eight depicts, that a given pixel is damaged.

Greater resolution would allow to show more details in an image but it would also lead to a greater size of CA, which has a negative impact on both the time needed to image reconstruction and searching rules. In our experiments we consider CA with Moore neighbourhood. The example of a cell with two defective pixels is illustrated in the Fig. 2.

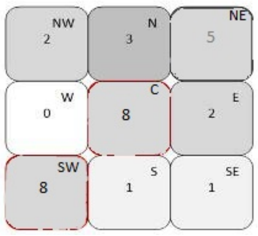


Fig. 2. An example of the Moore neighbourhood containing two cells representing damaged pixels

Let us consider Moore neighbourhood. For nine possible states of a cell, there exist $9^9 = 387420489$ possible neighbourhood states. It means that the number of possible rules equals to $9^{387420489}$. Searching the best rule is impossible in practise because each possible rule should be examined in image reconstruction, which is very time consuming. Less time demanding problem relates to von Neumann neighbourhood, which consists of five cells. To overcome the problem with finding a good rule in such a large search space we are going to employ GA, which is known as a good tool widely used in many combinatorial problems.

4.1 Image Coding Scheme

While GA maintains a population of individuals - CA rules, their coding scheme must be defined. An individual must depicts a state of a cell for each possible neighbourhood. Let us analyze an example of a neighbourhood and a part of an indvidual presented on figure 3.

CA rule is represented by the vector containing values in the range between 0 and 8 representing a color tone. Nine cells of Moore neighbourhood are usually described by directions: North-West (NW), North (N), North-East (NE), West (W), Central (C), East (E), South-West (SW), South(S), South-East(SE). Using this convention a neighbourhood can be interpreted as a vector containing numbers representing a color of pixels in this neighbourhood. For the neighbourhood

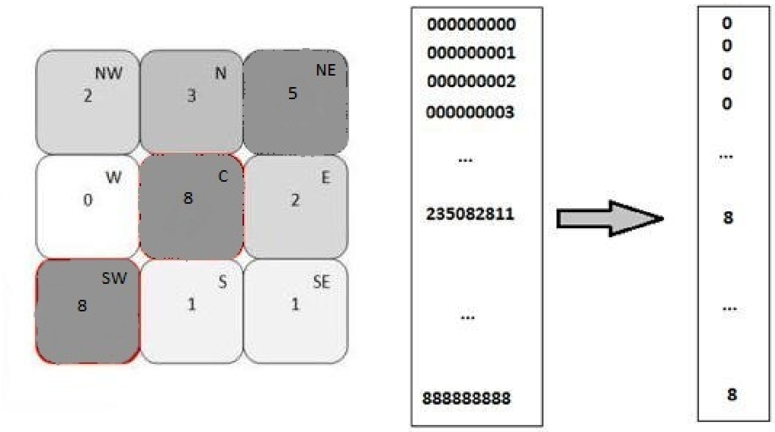


Fig. 3. An axample of the neighbourhood (on the left) and the part of the rule - the individual of the GA (on the right)

presented in the figure 3 such a vector will contain values read from left to right and from top to bottom e.g. 235082811. For each possible neighbourhood there is also given a value which central cell of a neighbourhood will have in the next time step of CA.

4.2 Fitness Function

Fitness of an individual is calculated according to Eg. 1

$$f = (n_c - (n_d + n_i)^t)/n, \tag{1}$$

where n_c is the number of cells with the proper state, n_d means the number of cells corresponding to damaged pixels, n_i depicts the number of cells with wrong state but not damaged pixel, t is a coefficient and n is the number of all cells. The objective of GA is to find such a rule that together with CA will allow to obtain reconstucted image with maximum number of pixels with original color.

At each generation of the GA, individuals are evaluated and the obtained fitness value corresponding to each one is used by the selection operator. It selects individuals from the current population to the temporary one proportionally to their fitness value and the obtained individuals take part in genetic modifications through crossover and mutation operators.

4.3 Training Phase

The objective of the training phase is to find rules by the GA. The incomplete image is an initial configuration of the CA. GA starts with a population of randomly generated rules. In the next step all individuals are evaluated against an image with p% randomly damaged pixels. Next, a rule is applied to the CA for T time steps and the obtained image is used to calculate fitness value of the rule according to the Eg. 1. Once we have evaluated individuals GA starts to improve them through the application of selection, crossover and mutation operators. The pseudocode of the GA is presented as Algorithm 1.

Algorithm 1. GA for searching CA rules - learning mode

```
1:  begin
2:      present an original image and create the corresponding CA;
3:      generate the initial population of CA rules of size $P$;
4:      generate an image with $p$% damaged pixels of the original image;
5:      for each rule in the population do
6:          run CA during $T$ time steps;
7:          compute the fitness function value;
8:      end
9:      for $i := 1$ to $G$ (generations) do
10:         randomly choose $P$ rules from the current population, with replacement;
11:         divide $P$ chosen rules into disjoint pairs;
12:         cross each pair by means of one point crossover;
13:         mutate offsprings with the probability $p_m$;
14:         generate an image with $p$% damaged pixels of the original image;
15:         for each rule in the population do
16:             run CA during $T$ time steps;
17:             compute the fitness function value;
18:         end
19:     end
20:     choose the best individual from the population as the result;
21: end
```

4.4 Testing Phase

After the best rule have been obtained from GA, in the next step it is examined how well CA is able to recover an original image from its distorted version. Moreover, we want check if the obtained rule has generalization feature. It would mean that a rule corresponding to one image is able to perform image reconstruction for the other images of the same class. While we are concerned with the images presenting people faces, the other images relate to different person. Testing mode of the GA is performed according to Algorithm 2.

First, we have obtained three rules in the learning mode of the GA. They were evolved to reconstruct images $IMG1$, $IMG2$ and $IMG3$. Next, three CA are created that correspond to the same image $IMG1$ with p% damaged pixels. The first CA is governed by the rule, which was assigned to the image $IMG1$, the second CA is performed using the rule assigned for the image $IMG2$ and

Algorithm 2. GA for searching CA rules - testing mode

```
1: begin
2:     get the best three rules $rule\_IMG_1$, $rule\_IMG_2$ and $rule\_IMG_3$ from the
       GA that were evolved to images: $IMG1$, $IMG2$ and $IMG3$, respectively;
3:     create three CAs: $CA1$, $CA2$ and $CA3$ that correspond to the same image
       $IMG1$ with damaged pixels;
4:     while image quality increases do
5:         perform one step of $CA1$ using $rule\_IMG_1$;
6:         perform one step of $CA2$ using $rule\_IMG_2$;
7:         perform one step of $CA3$ using $rule\_IMG_3$;
8:     end
9:     for each CA do
10:        compute the quality function value;
11:    end
12: end
```

the third CA is run with the rule assigned to reconstruct the image $IMG3$. For each CA, change the state of the CA until quality of the image that is reflected by the CA is not getting better. At the end of the testing mode, for each image calculate how many pixels were restored correctly. This value, denoted as f_1, is computed according to Eq. 2

$$f_1 = n_c/n, \tag{2}$$

where n_c is the number of correctly recovered pixels and n depicts the number of all pixels.

5 Experimental Results

5.1 Training

Three images denoted as $T0$, $T2$ and $T3$ were used in the experiments. For each image, we tested the performance of the GA for a case of 70% damaged pixels within an image.The maximal number of steps during which CA has to converge to a desired original image was set to 40. Experimental results show, that even for 20 time steps CA converges. The parameters of the GA were the following: population size = 50, crossover probability = 0.7 and mutation probability = 0.02. The searching process was conducted for 50 iterations of the GA. The Fig. 4 presents the performance of the GA.

We can see that for images $T2$ and $T3$ fitness value for randomly generated rules was between 0.35 and 0.45. In the evolutionary process GA was able to find better rules and in about 25 generation fitness value of the best rules equals to 0.6. The performance of the GA for image $T0$ was better than the performance of the GA for the two previous images. In the next 20 iterations of the GA, quality of found rules increased to about 0.77 value. Getting stuck in local minimum

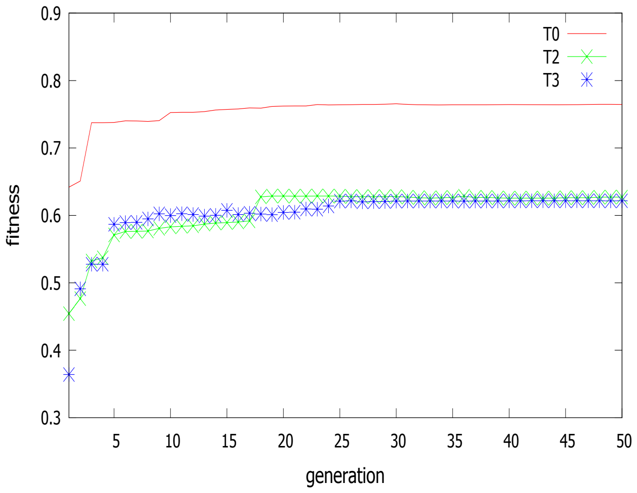


Fig. 4. Performance of GA in searching rules

around 25 iterations may be due to the vast space of solutions. Figure 5 and 6 present how images are reconstructed in 1, 10 and 20 iterations of the GA and in 1, 5 and 10 time steps of the CA.

Both figures show the process of reconstructing images of two different people. Three rows of images correspond to the first, tenth and twentieth iteration of the GA. Three columns of images depicts 1, 5 and 10 step of CA. In both figures it is shown that the quality of rules found by GA increases in subsequent iterations. In the 20th iteration of the GA and the first step of the CA the number of defective pixels is very small and the quality of the images in the following steps of CA increases.

5.2 Testing

The objective of the testing phase of the experimental study was to examine if the obtained rule is able to reconstruct an image with damaged pixels. We were also interested if the other rules obtained for different images but of the same class are able to reconstruct an image. For this purpose three CAs were created for the same image. These CAs were performed using three different rules and at the end quality values were calculated according to Eq. 2. Figure 7 presents an image with defective pixels, which was used as input for three CAs.

Figure 8 presents images obtained from three CAs using three rules obtained from three various images. Images within the first row were obtained by an application of the rule dedicated to restore such an image. The images within the two next rows have been restored using rules dedicated for the other, different images. The calculated quaility value for images within the first, second and third row equals to 0.87, 0.82 and 0.85 respectively. It means that images have been reconstructed with 87%, 82% and 85% correct pixels. These results show

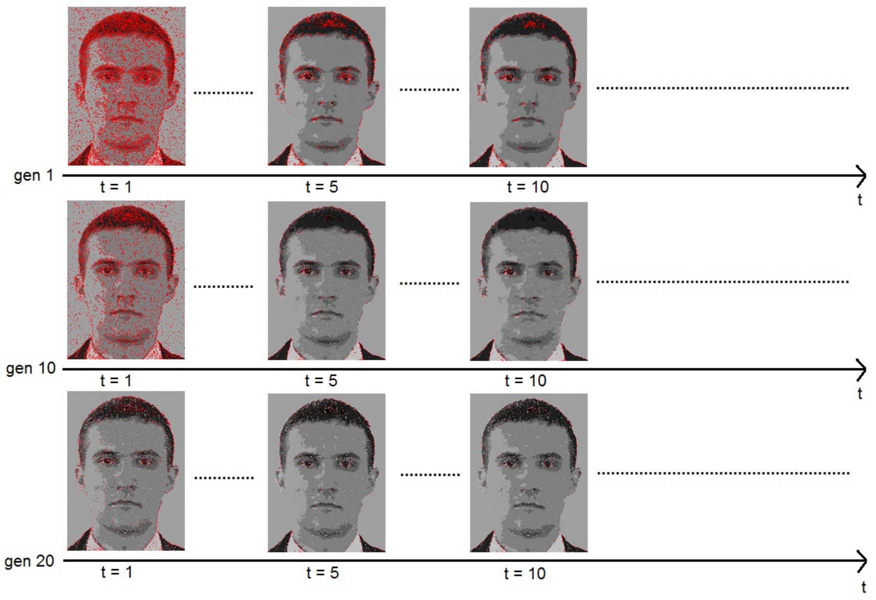


Fig. 5. Reconstruction of $T0$ image in the training phase

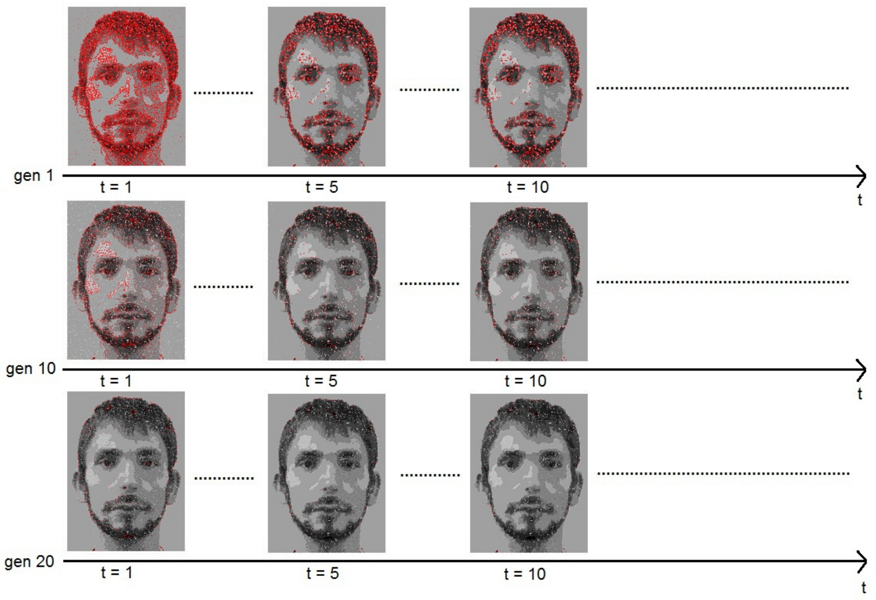


Fig. 6. Reconstruction of $T2$ image in the training phase

Fig. 7. Input image for testing CA

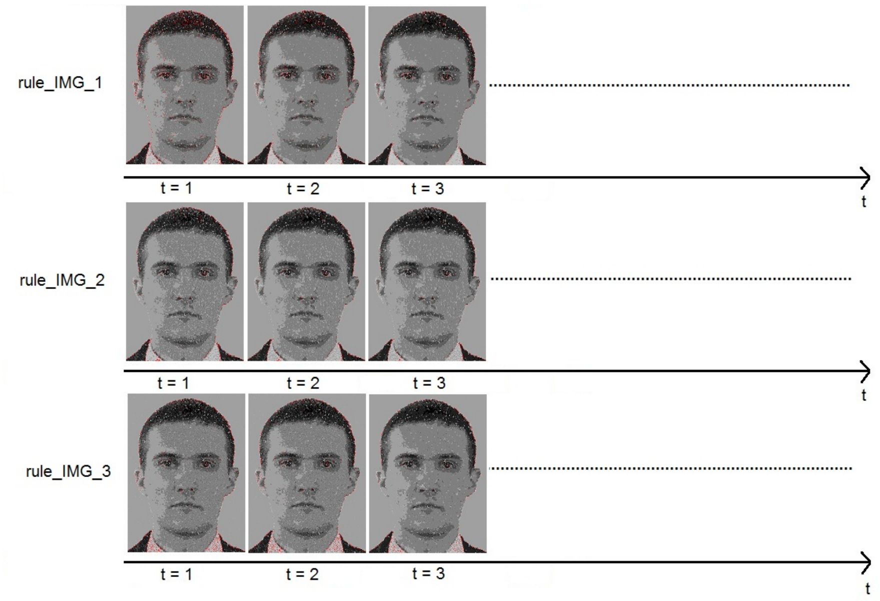


Fig. 8. Result of image reconstruction performed by CA

that the rule $rule\_IMG\_1$ assigned to the reconstructed image allows to restore the image with high quality. Moreover, despite the fact that the other two rules were assigned to the different images, their application allow to restore the image with nearly the same quality of rule $rule\_IMG\_1$. Such a result proves that there exists a subset of rules that are enough to restore a number of images, if these images are similar.

6 Conclusions

A new approach to the image reconstruction problem based on an application of GA for searching rules for CA has been shown in the paper. Our experimental results show that GA finds such good rules for CA, that even images with large number of damaged pixels can be restored to near the original image, which allows to recognize a face presented on image. Moreover, we proved that a rule found by GA possess generalization feature. It means, that it can be used to reconstruct images of the same class that the image, for which the rule was evolved by the GA.

The performance of the GA for searching rules and the CA for restoring original image from very low quality image is very promising. We believe, that this work could be initial research that would allow to take the new approach into practise and take advantage of it by commercial software.

References

1. Bandini, S., Vanneschi, L., Wuensche, A., Shehata, A.B.: A neuro-Genetic Framework for Pattern Recognition in complex Systems. Fundamenta Informaticae 87(2), 207–226 (2008)
2. Banham, A., Katsaggelos, A.: Digital image restoration. IEEE Signal Processing Magazine 14, 24–41 (1997)
3. Breukelaar, R., Bäck, T.: Evolving transition rules for multi dimensional cellular automata. In: Sloot, P.M.A., Chopard, B., Hoekstra, A.G. (eds.) ACRI 2004. LNCS, vol. 3305, pp. 182–191. Springer, Heidelberg (2004)
4. Das, R., Crutchfield, J., Mitchell, M.: Evolving globally synchronized cellular automata. In: Proceedings of the 6th International Conference on Genetic Algorithms, pp. 336–343 (1995)
5. Fawcett, T.: Data mining with cellular automata. ACM SIGKDD Explorations Newsletter 10(1), 32–39 (2008)
6. Hernandez, G., Herrmann, H.: Cellular automata for elementary image enhancement. Graphical Models and Image Processing 58(1), 82–89 (1996)
7. Mitchell, M., Hraber, P., Crutchfield, J.: Revisiting the edge of chaos: Evolving cellular automata to perform computations. Complex Systems 7, 89–130 (1993)
8. Popovici, A., Popovici, D.: Cellular automata in image processing. In: Proceedings of the 15th International Symposium on Mathematical Theory of Networks and Systems (2002)
9. Rosin, P.L.: Training Cellular Automata for Image Processing. IEEE Transactions on Image Processing 15(7), 2076–2087 (2006)
10. Slatnia, S., Batouche, M., Melkemi, K.E.: Evolutionary cellular automata based-approach for edge detection. In: Masulli, F., Mitra, S., Pasi, G. (eds.) WILF 2007. LNCS (LNAI), vol. 4578, pp. 404–411. Springer, Heidelberg (2007)
11. Swiecicka, A., Seredynski, F., Zomaya, A.Y.: Multiprocessor scheduling and rescheduling with use of cellular automata and artificial immune system support. IEEE Transactions on Parallel and Distributed Systems 17(3), 253–262 (2006)
12. Wolfram, S.: A New Kind of Science. Wolfram Media (2002)
13. Xiao, X., Shao, S., Ding, Y., Huang, Z., Chou, K.-C.: Using cellular automata images and pseudo amino acid composition to predict protein subcellular location. Amino Acids 30, 49–54 (2006)

Analysis and Application of the Pedestrian Permeation through the Crowd via Cellular Automata

Kenichiro Shimura[1,2], Giuseppe Vizzari[2], Stefania Bandini[2], and Katuhiro Nishinari[1]

[1] Research Center for Advanced Science and Technology, The University of Tokyo, 4-6-1 Komaba, Meguro-ku, Tokyo 153-8904 Japan

[2] Department of Informatics Systems and Communication, The University of Milano Bicocca, Viale Sarca 336 - U14, 20126 Milano, Italy

shimura@tokai.t.u-tokyo.ac.jp

Abstract. This paper studies pedestrian interaction where pedestrians permeate thought crowds in areas such as train stations and shopping centres. Pedestrian simulation model is created by use of Cellular Automata (CA) and various analyses are carried out by means of pedestrian jam. This study further attempts to provide a solution to crowd management. The results show that congestion can be eased by employing intervals between pedestrians. In order to obtain the best possible interval values, a utility function is introduced. The calculated interval with maximum utility provides the optimal solution to ease pedestrian congestion and jam. Furthermore, taking pedestrian interval as a control variable, this Model is applied to train arrival at terminal stations. The result of the simulation herein suggests a solution by implementing a delayed opening of train doors.

Keywords: Cellular Automata, pedestrian, crowd management, jam absorption.

1 Introduction

Pedestrians passing through crowds are often seen in daily life. This study aims to provide insights for modelling and analyzing interactions between pedestrians and crowds by use of Cellular Automata (CA). CA is defined in a discretised time and space where the dynamics is described by microscopic local interaction rules, wherein the movement of particles in a discrete time step is expressed by the relationship between the current state of particles and its surrounding conditions. The movement of particles is a stochastic process whereby the probability of change of the particle position and/or the state is defined by the transition probability. Thus, the macroscopic dynamics is represented as a result of time evolution. Extensive study on pedestrian dynamics has been carried out in the last decade aiming for various social applications such as evacuation and crowd control. [1-3]. Further, a theoretical study on the interaction between pedestrians and crowds was conducted by the author [4] and demonstrated that the dynamics can be expressed by one-dimensional Asymmetrical Simple Exclusion Process (ASEP) and resulting Burgers' equation.

The model proposed herein is applied to the terminal station where platforms and concourses exist, in which one end of the platform connects to the concourse. Platforms and concourses have different functions where the former is for accessing the train and the latter for multi-purposes including, platform and station access, waiting, ticketing, and shopping. A crowd is defined in this paper as people in the concourse moving with various speed and directions. On the other hand, pedestrians are defined as a stream of people moving towards the same direction with common purpose, *i.e.,* stream of people getting off the train, moving toward the exit. Depending on the time of the day, density of the crowd in the concourse will create jam in the platform with pedestrians queuing to enter the concourse. The objective of this paper is to analyze the behaviour of this jamming phenomenon.

2 Modelling

The square lattice is divided into the pedestrian and crowd area as illustrated in Fig. 1. The divided lattices are adjacent to each other abling pedestrians to move from the pedestrian area into the crowd area, accessing the Exit. The pedestrian motion is uni-directional, starting from a certain position within the pedestrian area then, penetrate into the crowd area. There is also a possibility of a growing queue in the pedestrian area at the boundary of the crowd area.

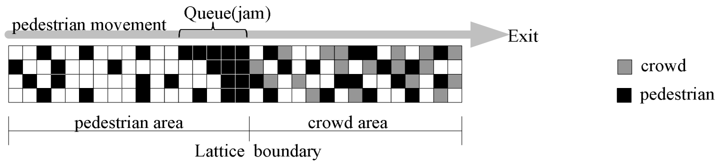


Fig. 1. The lattice configuration for the model, where the pedestrians and crowds are black and grey, respectively

In microscopic view, the dynamics of individuals are characterized by its direction and speed. Large variation on these characteristics results in fairly random macroscopic behaviour. Thus, the dynamics of the crowd is simplified to a random replacement. This implementation expresses the effect of the crowd as a white noise to the pedestrians. The degree of the effect is defined by the crowd density ρ, where higher density restricts more of the pedestrian's mobility inside the crowd. The random replacement however gives the occupying probability equivalent to the density to every cell at every time step. Two types of models are implemented for the pedestrian dynamics. One is a simple ASEP where the pedestrians walk along a straight line without changing lanes and the transition rule is illustrated in Eq. 1.

The other rule is that the pedestrians overtake others as shown in Eq. 2 while in the ASEP, for a particular particle occupying the cell, if the cell in front of it is occupied by another, then the particle cannot move.

$$n_i^{t+1} = Pn_{i-1}^t(1-n_i^t-m_i^t)+n_i^t n_{i+1}^t + n_i^t m_{i+1}^t + (1-P)(1-n_{i+1}^t - m_{i+1}^t)n_i^t \tag{1}$$

Where $m_i^t = (1-n_i^t)\rho$

$$\begin{aligned} n_{i,j}^{t+1} &= (1-n_{i,j}^t - m_{i,j}^t)\big(Pn_{i-1,j}^t + (1-P)(1-n_{i-1,j}^t - m_{i-1,j}^t) \\ &\big(n_{i-1,j+1}^t(1-n_{i-1,j-1}^t)(m_{i,j+1}^t + n_{i,j+1}^t) + n_{i-1,j-1}^t(1-n_{i-1,j+1}^t)(m_{i,j-1}^t) \\ &+ \frac{1}{n_{i-1,j-1}^t + n_{i-1,j+1}^t} n_{i-1,j+1}^t n_{i-1,j-1}^t + (m_{i,j+1}^t m_{i,j-1}^t + n_{i,j+1}^t n_{i,j-1}^t)\big)\big) \end{aligned} \tag{2}$$

Where $m_i^t = (1-n_i^t)\rho$

Eq. 1 and Eq. 2 shows the occurrence probability of the site *i,j* is occupied by a pedestrian at time *t*+1 in terms of the states of its surrounding cells at time *t*. The notations are as follows: n^t_{ij} and m^t_{ij} denotes the occurrence probability of a pedestrian and crowd, respectively at time *t* in site *i,j*. *P* denotes the transition probability of the pedestrians which can be used as the implementation of walking speed. The crowd density is denoted as ρ. At the lattice boundary, the queuing theory is applied. When the pedestrian arrives to the boundary and the crowd area is full where the pedestrians do not have space to enter, a queue is created. The queue is expressed as a simple mass balance of those pedestrians entering and exiting the queue. Therefore, the pedestrian path has three sections: 1, pedestrian area described by ASEP; 2, queue; and 3, crowd area described by Eq.1 or Eq.2.

3 Simulation and Discussions

The behaviour of the pedestrian is highly affected by the density of the crowd. Fig. 2 schematically illustrates the simulation condition. The figure expresses the pedestrians approaching the crowd area with interval *s*, a queue at the boundary, and pedestrians entering the crowd and coming out from the Exit.

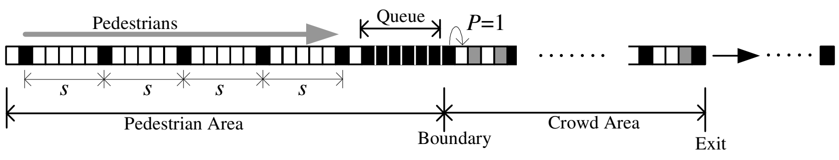


Fig. 2. The simulation condition. The pedestrian interval *s*.

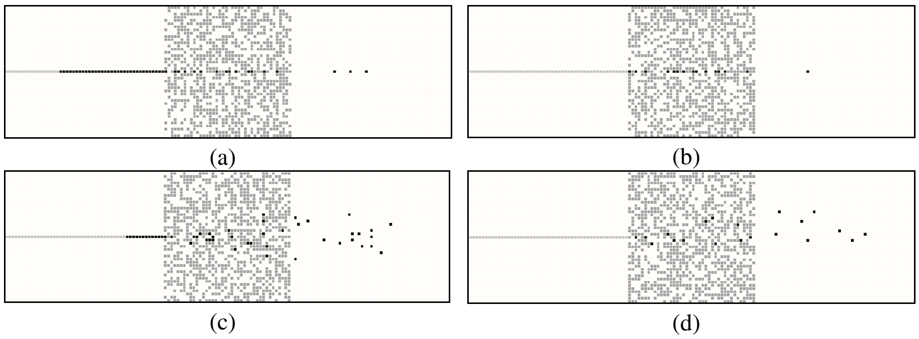


Fig. 3. The image of simulation output after 100 steps for (a) $s = 1$, without overtake, (b) $s = 6$, without overtake, (c) $s = 1$, with overtake., (d) $s = 4$, with overtake

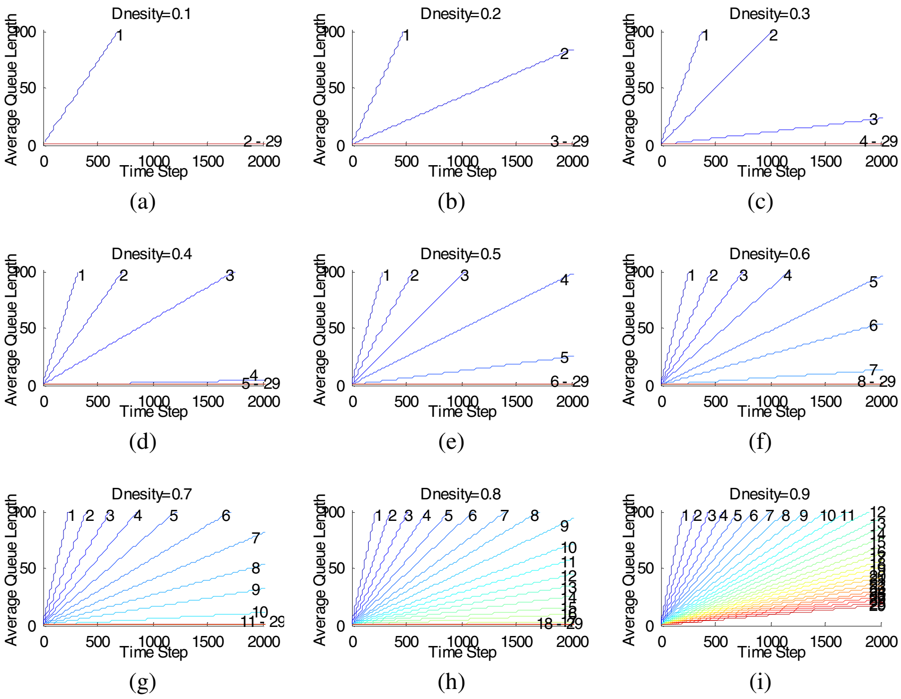


Fig. 4. The queue length with respect to time for various pedestrian's interval. (a) Crowd density = 0.1, (b) 0.2, (c) 0.3, (d) 0.4, (e) 0.5, (f) 0.6, (g) 0.7, (h) 0.8, (i) 0.9. Simulated for the model without overtake. The number on each graph shows pedestrian's interval.

In order to analyse the behaviour of the queue at the boundary, the simulations are carried out for various crowd densities and various pedestrian intervals. Fig.3 shows the image of simulation output for various conditions when the crowd density is taken as 0.5. In the figure, the crowd is illustrated as grey and pedestrians as black. Fig. 3(a) and (b) are cases where pedestrian do not overtake others and pedestrian interval is 1 and 6 respectively. Whereas, Fig. 3(c) and (d) are the case where pedestrians overtake others and pedestrian interval is 1 and 4 respectively. Fig. 3(a) and (c) shows a growth of queue despite Fig. 3(b) and (d) do not. This feature suggests that the pedestrian's interval is greatly affects queue length.

Further, change of queue length with respect to time for various crowd densities is calculated. Fig.4 shows the result where the density of crowd varies from 0.1 to 0.9 and pedestrian interval is varies from 1 to 29. The pedestrian model inside the crowd is illustrated by Eq. (1). In general, there is certain threshold in the pedestrian interval where the queue do not grow with time. For example, when the crowd density is 0.2, the queue grows with time when the pedestrian interval is 1 and 2, on the other hand, queue length remains 0 for the case with intervals are larger than 3. This phenomenon is caused by the mass balance of the queue that the balance of the arrival rate to the queue and the exit rate from the queue. The queue grows when the arrival rate is larger than the exit rate and the queue shrinks when the arrival rate is smaller. In this system, the pedestrian interval affects the arrival rate while the crowd density affects the exit rate. The profile shows that longer interval is necessary for higher crowd density to eliminate the queue. This characteristic remains the same when the overtaking model is applied.

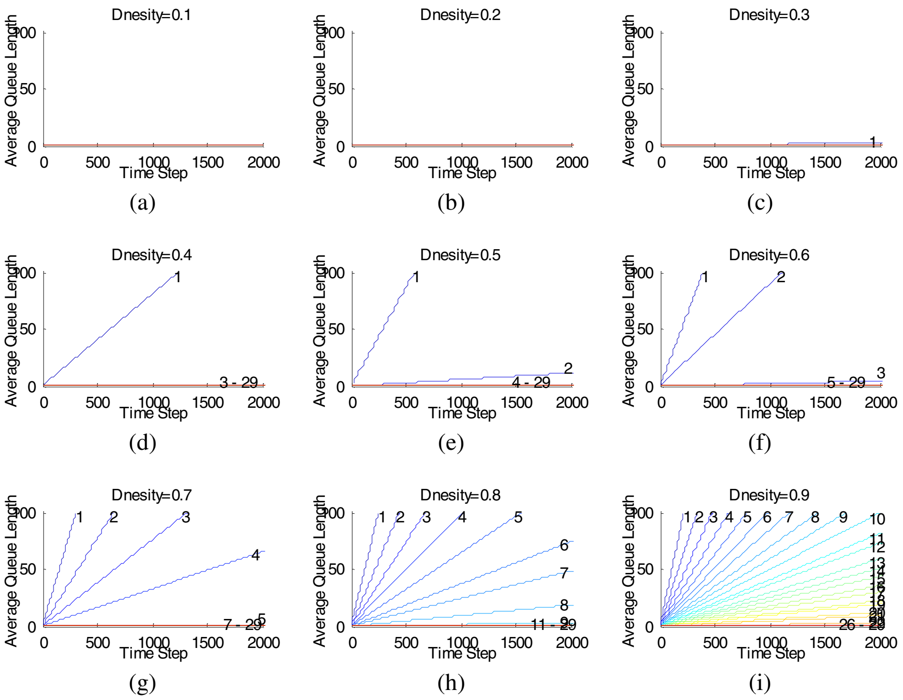


Fig. 5. The queue length with respect to time for various pedestrian's interval. All the conditions are the same as that for Fig. 4 except overtaking model *c.f.,* Eq. (2) is applied.

Fig.5 shows the calculation result of those simulated under the same conditions as that for Fig. 4 except the overtaking pedestrian model as shown in Eq. (2) is applied. The difference is that the threshold of the pedestrian intervals for the queue elimination is smaller in general. This is because the pedestrians gain more mobility inside the crowd thus the exit rate of the queue can be larger, and therefore higher arrival rate into the queue is accepted. The difference of the threshold between two models becomes larger as crowd density increase, *e.g.,* when crowd density is 0.7, for eliminating the queue, the interval threshold reduced to 7 in overtaking model whereas 11 in non-overtaking model.

It is also seen from Fig.4 and Fig 5 that the queue grows linearly with time. The gradient of queue length with respect to time can be calculated by least square method. The gradient shows the growth rate of the queue, this means that when this value is larger, then the queue grows faster. Fig.6 illustrates the queue growth rate against pedestrian's intervals. The figure shows in general that the queue growth rate decays with increasing intervals. From the point of view of crowd control, it is of great importance to understand this behaviour since pedestrian interval is the effective control variables under some conditions.

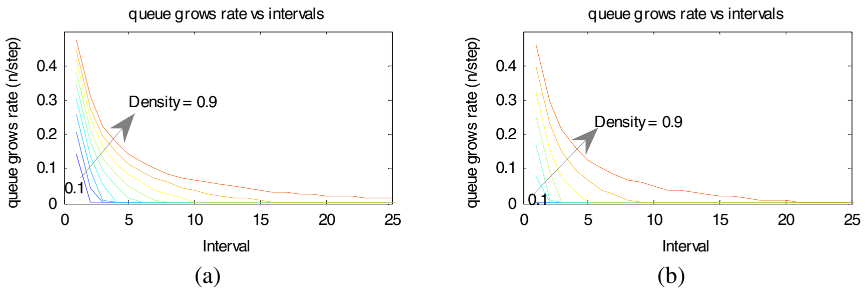
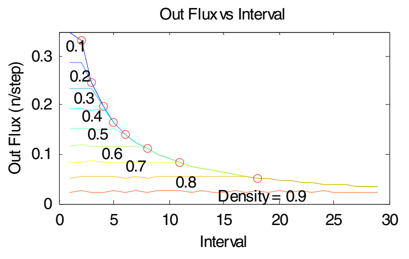


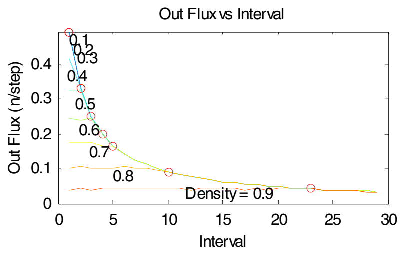
Fig. 6. The queue growth rate with respect to pedestrian intervals for various crowd densities of 0.1 to 0.9. (a) Model without overtake *c.f.,* Eq. (1). (b) Model with overtake *c.f.,* Eq. (2).

Further analyses are carried out in terms of the pedestrian flux and necessary time to handle certain number of pedestrians. Fig.7 illustrates the flux of pedestrians outgoing from the crowd plotted against pedestrian's interval for various crowd densities of the range 0.1 to 0.9. Fig.7 (a) shows the result of pedestrian model without overtake and (b) shows the pedestrian model without overtake. In the figure, the circle plot shows the threshold of pedestrian intervals where the growth rate of queue become zero. It is seen that the profile of the flux consist of two parts: 1, the plateau region at smaller intervals, and 2, asymptotic region at higher intervals. The plateau region occurs for the interval range smaller than the threshold. In this situation, there are growing queues at the entrance to the crowd area in which pedestrians are waiting to enter the area. The pedestrian at the front of the queue enters the crowd area whenever one finds space to enter. When the pedestrian enters the crowd area and proceeds forward, such movement creates a space behind. At this moment another pedestrian is at the front of the queue waiting for his chance to enter.

This is the pedestrian balance at the boundary and affects the exit rate of the queue. On the other hand, the growth rate at the end of the queue depends on the pedestrian interval values. This means that the flux is rate-limited by the crowd density. As shown, while increasing the pedestrian interval, the queue disappears at certain threshold point. Thereafter the threshold, the flux is determined by the pedestrian intervals. Thus, when a pedestrian arrives at the entrance of the crowd area, one can enter without queuing, having the flux determined by this arrival rate. As the pedestrian interval become longer, the flux is dramatically reduced although the queue disappears. In this region the flux is rate-limited by the pedestrian intervals. This phenomenon can be seen more clearly when the required time for certain number of pedestrians to cross through the crowd is calculated. Fig. 8 illustrates the plots of required times to handle 1000 pedestrians against pedestrian interval for various crowd densities of the range of 0.1 to 0.9. Similarly to those profiles for fluxes, the required time also shows the plateau regions. This result shows that the total required time do not vary while there is a queue, thus, when intervals increases more time is required to handle the pedestrians.

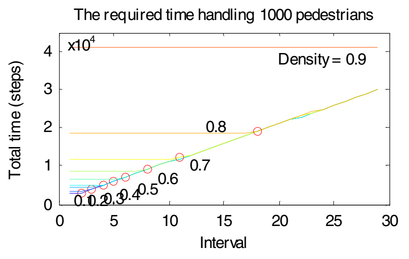


(a)

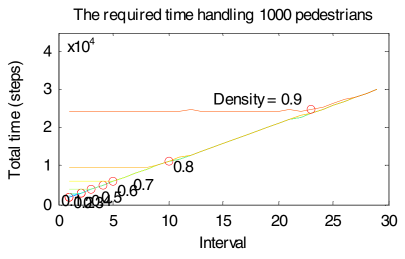


(b)

Fig. 7. The flux of pedestrians exiting from the crowd. Where crowd densities are 0.1 to 0.9. (a) Model without overtake *c.f.,* Eq. (1). (b) Model with overtake *c.f.,* Eq. (2). The circles shows the threshold of pedestrian's interval where the queue growth rate become nearly zero.



(a)



(b)

Fig. 8. The required time handling 1000 pedestrians for various crowd densities which the number is superimposed in the figure. (a) Model without overtake *c.f.,* Eq. (1). (b) Model with overtake *c.f.,* Eq. (2).

It is also important to note that the asymptotic profile seen in the Fig.7 and Fig. 8 suggest that there is a maximum flow limit. As it is seen from the figure, the asymptotic profile is independent of the crowd density. Starting from various densities, the maximum achievable flux or handling time solely depends on the pedestrian interval. This result is an important point that the pedestrian interval can be a critical control variable for crowd management. Form the point of view of crowd management, here in this work, in order to find the most likelihood value of interval to achieve the maximum flux and least queue length, the utility function is introduced as shown Eq. (3). The utility function *U* is defined by the product of the outgoing flux and the value that queue growth rate is subtracted by unity.

$$U = \left((1-\dot{q})(\dot{o})\right)^2 \tag{3}$$

Where $\dot{q}$: the queue growth rate (*c.f.* Fig.6)

$\dot{o}$: the outgoing flux. (*c.f.* Fig.7).

Fig. 9 illustrates the plot of the utility function. As it is seen from the figure, in general, a single peak appears. The pedestrian interval at the maximum utility gives the solution for "high pedestrian flux" and "low queue growth rate". The pedestrian intervals for maximum utility obtained by Eq.(3) for various crowd densities for two models are illustrated in Table 1. It is noted that the queue growth rate takes non-zero minimum value at maximum utility. This is because the longer interval has the effect of reducing the utility. In the event the crowd management policy do not allow any queues, then the value of pedestrian interval which is one unit larger than that for maximum utility should be taken to achieve the highest utility within the allowable condition.

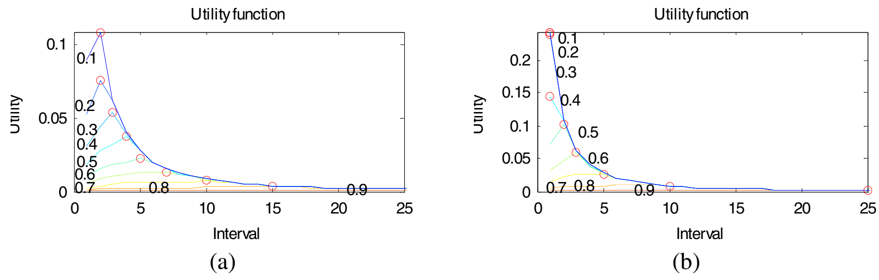


Fig. 9. Plot of the utility function shown in Eq. (3), where (a) Model without overtake *c.f.,* Eq. (1). (b) Model with overtake *c.f.,* Eq. (2). The circle on the plot shows the points where the utility become maximum.

Table 1. The pedestrian interval for maximum utility

Crowd density	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Interval (No overtake model)	2	2	3	4	5	7	10	17	n/a
Interval (Overtake model)	1	1	1	2	3	3	5	10	25

4 Application to the Crowd Control at the Station

The CA model is applied to the terminal station situation aiming to find a solution to ease pedestrian jam and congestion on the platform. Fig. 10 illustrates the arrival situation at the terminal. Pedestrians de-train and walk to the concourse. Assuming to have 10 coaches with each holding 100 passengers, the simulation is carried out varying the crowd density of the concourse. Moreover, for the pedestrian simulation, the model with overtaking, *c.f.* Eq. (2) is applied since it is more realistic.

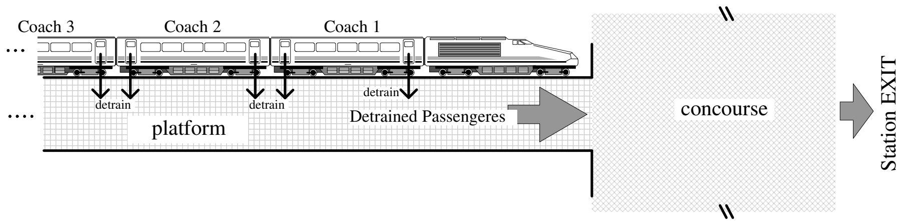


Fig. 10. The situation considered for the simulation

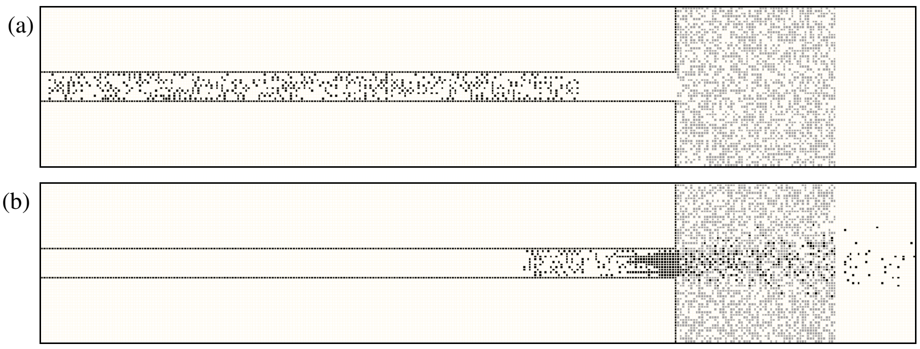


Fig. 11. The platform simulation without pedestrian control. (a) Initial State, (b) Transient state with pedestrian jam at the entrance to the concourse. Where the crowd density is 0.5.

Fig. 11 illustrates the simulation result when crowd density is 0.5 and pedestrians are randomly allocated on the platform without any interval. This simulation condition assumes that passengers de-train all at once from each coach. Fig. 11(a) shows the initial condition and Fig. 11(b) show the transient behaviour when there is a pedestrian jam at the end of the platform. Here again, the pedestrians are queuing to enter the concourse to pass through to the other side since the crowd in the concourse is obstructing pedestrian flow. Contrary to this, Fig. 12 illustrates the result when some interval exists between groups of pedestrians. The number of pedestrians in each group is set at 100, equalling the capacity of each coach. Fig. 12 (a) is the initial condition and Fig. 12 (b) and (c) shows the transient conditions. Fig12 (b) shows that a group of the pedestrians are going into the concourse while the next group of passengers are approaching. Fig. 12 (c) shows that the next group of passengers are arriving at the end of the platform. At this moment, the jam created by the previous group of passengers is solved. Therefore, there is no acculturation of queues. This is the basic idea of crowd management by separating the stream of pedestrians and making space between groups of passengers. If the door is opened one by one with some delay, *i.e.,* interval of several seconds, then the condition of Fig. 12 is created. This condition will make the passengers to wait inside the train for longer periods after arrival, but is an effective way to control congestion in the station.

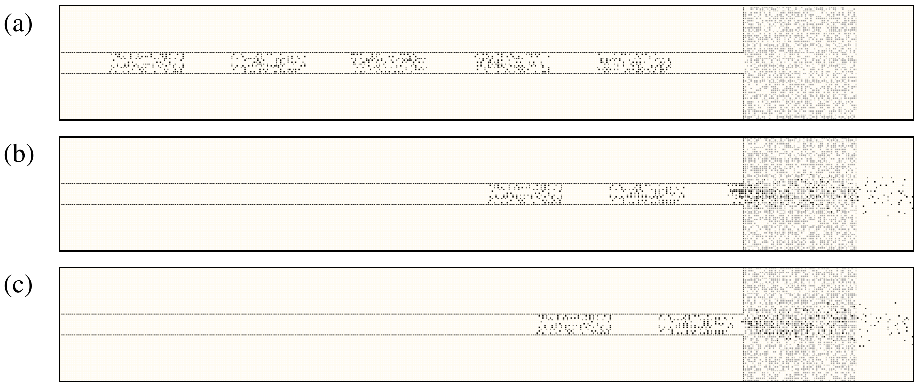


Fig. 12. The platform simulation with pedestrian control. (a) Initial State, (b) and (c) Transient state with pedestrian jam at the entrance to the concourse. Where the crowd density is 0.5.

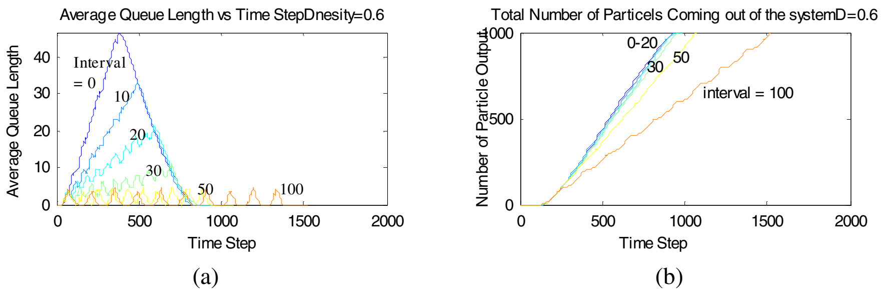


Fig. 13. (a) average queue length vs. time (b) total number of passengers exiting from the station. Where the total passenger is 1000 and crowd density is 0.6.

Fig. 13 (a) shows the average queue length with respect to time. An oscillation is seen in the plot at higher interval values. The oscillation is caused by the decrease in queue length while the interval is as seen in Fig. 12 (b) and (c). During the intervals there is no pedestrian arriving to the end of the queue thus the queue length decreases until the next group of pedestrians arrive. The sharp drop on the queue length after the maxima on the plot means that only passengers in the queue remain on the platform.

Thus, the queue length linearly drops to zero since there is no more passengers arriving. Fig. 13 (b) shows the total number of passengers exiting from the station. It is seen that when the interval is large, the plot shape resembles a series of steps. The step is created because the interval is too large that there are moments when there is no queue at the boundary.

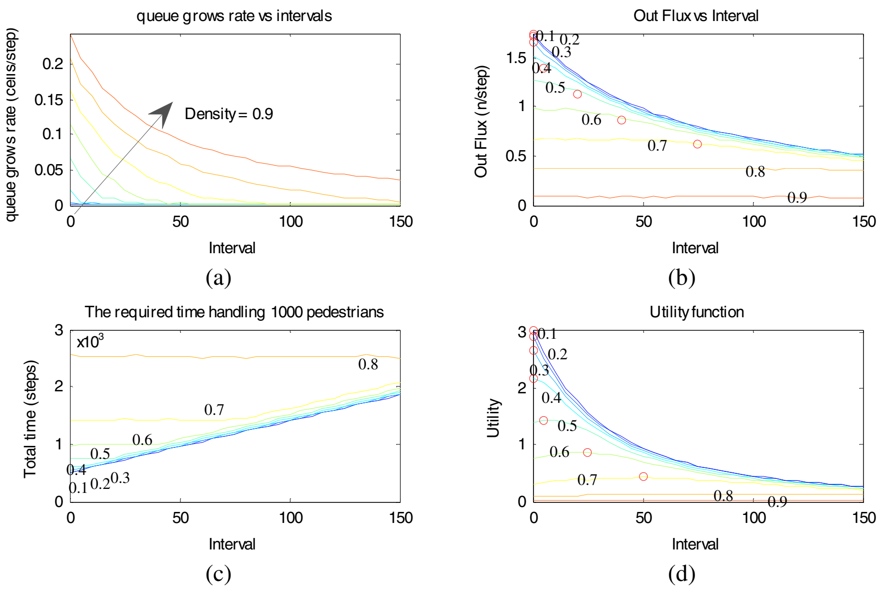


Fig. 14. (a) The queue growth rate with respect to pedestrian intervals. (b) The flux of pedestrians exiting from the crowd. (c) The required time handling 1000 pedestrians. (d) Plot of the utility function. For all figures, calculations are carried out for various crowd densities of 0.1-0.9 where the number is superimposed in the figure.

Fig. 14 (a), (b), (c) and (d) shows the queue growth rate with respect to pedestrian intervals, the flux of pedestrians exiting from the crowd, the required time handling 1000 pedestrians, and plot of the utility function, respectively. Fig. 14 (a), (b), (c) exhibits the basic characteristics as seen in Fig. 6 (b), Fig. 7 (b) and Fig. 8 (b), respectively. The pedestrian interval for maximum utility is marked on the Fig. 14 (d) and is summarised in Table 2. It is seen that for low crowd density below 0.4, the utility is maximum when there is no interval. It is also seen from Fig. 14 (a) that when the crowd density is below0.4, the queue growth rate is substantially small. Further, when increasing the crowd density, a longer interval is required. To illustrate, 25 cells interval is required for maximum utility when the concourse crowd density is 0.6. The common values for each cell are 0.4m square and walking speed of pedestrian is 1.3 m/s. Then a simple arithmetic gives 7.7 sec. Therefore, the interval to open each door should be eight seconds to manage pedestrian jam. This idea can be utilized for general situation in which pedestrians cross through crowds as in a shopping mall, where the escalator or lift is at one end and the entrance is at another end of the mall. In reality, there are much more considerable conditions but this result indicates that the pedestrian jam can be controlled or eliminated by relatively small effort.

Table 2. The pedestrian's interval for maximum utility

Crowd density	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Optimal Interval	0	0	0	0	5	25	50	N/A	N/A

5 Conclusion

The pedestrian crowd interaction model is introduced and various simulations are carried out for situation where a stream of pedestrians are entering into a crowd then permeate through to the other side. Depending on the crowd density, a jam occurs at the entrance of the crowd. The pedestrian jam can be controlled by implementing some time interval between the pedestrian entrance. This model is applied to the train terminal arrival scenario to shows that a jam on the platform can be eased by controlling the time interval for opening the door of each coach. The timing can be calculated by simulation for maximum utility. The results indicate that the terminal station congestion can be controlled by providing a short wait period in the train for passengers with minimum inconvenience for.

References

1. Kirchner, A., Klupfel, H., Nishinari, K., Schadschneider, A., Schreckenberg, M.: Simulation of competitive egress behavior: comparison with aircraft evacuation data. Physica A 324, 689 (2003)
2. Kirchner, A., Schadschneider, A.: Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. Physica A 312, 260 (2002)
3. Jiang, R., Wu, Q.S.: The moving behavior of a large object in the crowds in a narrow channel. Physica A 364, 457–463 (2006)
4. Shimura, K., Tanaka, Y., Nishinari, K.: Analysis of Obstacle Density Effect on Pedestrian Congestion Based on a One-Dimensional Cellular Automata. In: Bandini, S., Manzoni, S., Umeo, H., Vizzari, G. (eds.) ACRI 2010. LNCS, vol. 6350, pp. 513–522. Springer, Heidelberg (2010)

An Isotropic Optimum-Time FSSP Algorithm for Two-Dimensional Cellular Automata

Hiroshi Umeo, Keisuke Kubo, and Yusuke Takahashi

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 synchronization in cellular automata, known as firing squad synchronization problem (FSSP), 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 propose a new recursive-halving based optimum-time synchronization algorithm that can synchronize any rectangle two-dimensional (2D) arrays of size $m \times n$ with a general at one corner in $m + n + \max(m, n) - 3$ steps. The algorithm proposed is quite different from previous designs and it can be easily generalized to 2D arrays with a general at any position of the array. The algorithm is isotropic concerning the side-lengths of 2D arrays and its correctness is transparent and easily verified.

1 Introduction

Synchronization of large-scale networks is an important and fundamental computing primitive in parallel and distributed systems. The synchronization in ultra fine-grained parallel computational model of 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. We study the FSSP that gives a finite-state protocol for synchronizing cellular automata. The problem 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 propose a new recursive-halving based optimum-time synchronization algorithm that can synchronize any rectangle two-dimensional (2D) arrays of size $m \times n$ with a general at one corner in $m + n + \max(m, n) - 3$ steps. Specifically, we attempt to answer the following questions:

– Are there still any new FSSP algorithms for 2D arrays?
– Can we generalize the 2D FSSP algorithms to a generalized FSSP, where an initial general is located at any position of the array?

– What is an isotropic FSSP algorithm concerning side length of the given array?

The algorithm proposed is interesting in the following view points.

– The algorithm is quite different from previous designs and it gives a new insight into the 2D array synchronization mechanism.
– The 2D algorithm proposed is isotropic with respect to shape of a given rectangle array, i.e. no need to control the FSSP algorithm for longer-than-wide and wider-than-long input rectangles.
– The correctness of the algorithm is transparent and easily verified.
– The algorithm can be expanded to a generalized optimum-time FSSP solution, where an initial general is at an arbitrary position of a given array.

In Section 2 we give a description of the 2D FSSP and review some basic results on 2D FSSP algorithms. Section 3 defines the recursive-halving marking on 1D arrays. In Section 4 we present a new 2D FSSP algorithm. An implementation in terms of 2D finite-state cellular automaton is also presented for the optimum-time FSSP algorithm. It is shown that the algorithm can be generalized with respect to the general's position.

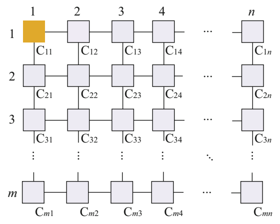


Fig. 1. A 2D cellular automaton

2 FSSP on Two-Dimensional Arrays

Figure 1 shows a finite 2D cellular array consisting of $m \times n$ cells. 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. 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 cell with quiescent neighbors is the quiescent state again. At time $t = 0$, the north-west corner cell $C_{1,1}$ is in the *fire-when-ready* state, which is the initiation signal for 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. The tricky part of the problem is that the same kind of soldier having a fixed number of states must be synchronized, regardless of the size $m \times n$ of the array. The set of states and next state function must be independent of m and n.

The problem was first solved by J. McCarthy and M. Minsky who presented a $3n$-step algorithm for 1D cellular array of length n. In 1962, the first optimum-time, i.e. $(2n-2)$-step, synchronization algorithm was presented by Goto [1962], with each cell having several thousands of states. On the other hand, several synchronization algorithms on 2D arrays have been proposed by Beyer [1969], Grasselli [1975], Shinahr [1974], Szwerinski [1982], Umeo, Hisaoka, and Akiguchi [2005], and Umeo, Nishide, and Kubo [2012]. It has been shown in Beyer [1969] and Shinahr [1974] independently that there exists no 2D cellular automaton that can synchronize any 2D array of size $m \times n$ in less than $m + n + \max(m, n) - 3$ steps. In addition they first proposed an optimum-time synchronization algorithm that can synchronize any 2D array of size $m \times n$ in optimum $m + n + \max(m, n) - 3$ steps. Shinahr [1974] gave a 28-state implementation. Umeo, Hisaoka and Akiguchi [2005] presented a new 12-state synchronization algorithm operating in optimum-step, realizing a smallest solution to the rectangle synchronization problem at present. As for the time optimality of the 2D FSSP algorithms, the following theorems have been shown.

Theorem 1$^{\text{Beyer [1969], Shinahr [1974]}}$**.** There exists no cellular automaton that can synchronize any 2D array of size $m \times n$ in less than $m+n+\max(m,n)-3$ steps, where the general is located at one corner of the array.

Theorem 2$^{\text{Umeo et al. [2005]}}$**.** There exists a 12-state cellular automaton that can synchronize any 2D array of size $m \times n$ at exactly $m+n+\max(m,n)-3$ steps, where the general is located at one corner of the array.

3 Recursive-Halving Marking

In this section, we introduce a marking scheme for 1D arrays referred to as *recursive-halving marking.* The marking schema prints a special mark on cells in a given cellular space defined by the recursive-halving marking. It is based on a 1D FSSP synchronization algorithm. The marking will be effectively used for constructing FSSP algorithms for 2D arrays operating in optimum-time.

Let S be a 1D cellular space consisting of cells C_i, C_{i+1}, ..., C_j, denoted by $[i...j]$, where $j > i$. Let $|S|$ denote the number of cells in S, that is $|S| = j - i + 1$. A center cell(s) C_x of S is defined by

$$x = \begin{cases} (i+j)/2 & |S|\text{: odd,} \\ (i+j-1)/2, (i+j+1)/2 & |S|\text{: even.} \end{cases} \qquad (1)$$

The recursive-halving marking for a given cellular space [1...n] is defined as follows:

Recursive-Halving Marking, RHM

```
Algorithm RHM($S$)
begin
    $S$:= [1...$n$];
    if $|S| \geq 2$ then
        if $|S|$ is odd then
            mark a center cell $C_x$ in $S$
            $S_L$:= [1...$x$]; $S_R$:= [$x$...$n$]
            $\mathbf{RHM}_{\mathrm{L}}(S_L)$; $\mathbf{RHM}_{\mathrm{R}}(S_R)$;
        else
            mark center cells $C_x$ and $C_{x+1}$ in $S$
            $S_L$:= [1...$x$]; $S_R$:= [$x+1$...$n$]
            $\mathbf{RHM}_{\mathrm{L}}(S_L)$; $\mathbf{RHM}_{\mathrm{R}}(S_R)$;
end
```

Left-Side Recursive-Halving Marking, RHM$_{\mathrm{L}}$

```
Algorithm RHM_L($S$)
begin
    $S$:= [1...$n$];
    while $|S| > 2$ do
        if $|S|$ is odd then
            mark a center cell $C_x$ in $S$
            $S_L$:= [1...$x$]; $\mathbf{RHM}_{\mathrm{L}}(S_L)$;
        else
            mark center cells $C_x$ and $C_{x+1}$ in $S$
            $S_L$:= [1...$x$]; $\mathbf{RHM}_{\mathrm{L}}(S_L)$;
end
```

The marking for the right-side half space, Right-Side Recursive-Halving Marking, RHM$_{\mathrm{R}}$, can be defined in a similar way.

Figure 2 (left) shows a space-time diagram for the marking. At time $t = 0$, the leftmost cell C$_1$ generates an infinite set of signals $w_1, w_2, ..., w_k$, .., each propagating in the right direction at $1/(2^k - 1)$ speed, where $k = 1, 2, 3, ...,$. The 1/1-speed signal w_1 arrives at C$_n$ at time $t = n - 1$. Then, the rightmost cell C$_n$ also emits an infinite set of signals $w_1, w_2, ..., w_k$, .., each propagating in the left direction at $1/(2^k - 1)$ speed, where $k = 1, 2, 3, ...,$. The readers can find that each crossing of two signals, shown in Fig. 2 (left), enables the marking at middle points defined by the recursive-halving. A finite state realization for generating the infinite set of signals above is a well-known technique employed in Balzer [1967], Gerken [1987], and Waksman [1966] for the implementations of the optimum-time synchronization algorithms on 1D arrays.

We have developed a simple implementation of the recursive-halving marking on a 13-state, 314-rule cellular automaton. In Fig. 2 (right) we present several snapshots for the marking on 42 cells. We have:

Lemma 3. There exists a 1D 13-state, 314-rule cellular automaton that can print the recursive-halving marking in any cellular space of length n in $2n - 2$ steps.

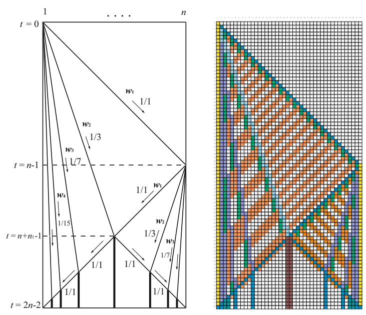


Fig. 2. Space-time diagram for recursive-halving marking on 1D array of length n (left) and some snapshots for the marking on 42 cells (right)

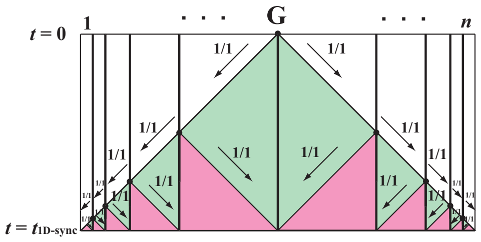


Fig. 3. Space-time diagram for synchronizing a cellular space with recursive-halving marking

The recursive-halving marking proposed played an important role in the classical WBG-type (Waksman [1966], Balzer [1967], and Gerken [1987]) FSSP algorithms. An optimum-time complexity $2n-2$ needed for synchronizing cellular space of length n in those algorithms can be interpreted as follows:

Let S be a 1D cellular space of length $n = 2n_1 + 1$, where $n_1 \geq 1$. The first center mark in S is printed on cell C_{n_1+1} at time $t_{1\mathrm{D-center}} = 3n_1$. Additional n_1 steps are required for the markings thereafter, yielding a final synchronization at time $t_{1\mathrm{D-opt}} = 3n_1 + n_1 = 4n_1 = 2n-2$. In the case $n = 2n_1$, where $n_1 \geq 1$, the first center mark is printed simultaneously on cells C_{n_1} and C_{n_1+1} at time $t_{1\mathrm{D-center}} = 3n_1 - 1$. Additional $n_1 - 1$ steps are required for the marking and synchronization thereafter, yielding the final synchronization at time $t_{1\mathrm{D-opt}} = 3n_1 - 1 + n_1 - 1 = 4n_1 - 2 = 2n - 2$.

$$t_{1\mathrm{D-center}} = \begin{cases} 3n_1 & |S| = 2n_1 + 1, \\ 3n_1 - 1 & |S| = 2n_1. \end{cases} \tag{2}$$

Thus, additional $t_{1\mathrm{D-sync}}$ steps are required for the synchronization for a cellular space with the recursive-halving marks:

$$t_{1\mathrm{D-sync}} = \begin{cases} n_1 & |S| = 2n_1 + 1, \\ n_1 - 1 & |S| = 2n_1. \end{cases} \tag{3}$$

In this way, it can be easily seen that any cellular space of length n with the recursive-halving marking and initially with a general on a single cell or two generals on adjacent center cells at time $t = 0$ can be synchronized in $t_{1\mathrm{D-sync}} = \lceil n/2 \rceil - 1$ optimum-steps. Thus we have:

Lemma 4. *Any 1D cellular space S of length n with the recursive-halving marking initially with a general(s) on a center cell(s) in S can be synchronized in $t_{1\mathrm{D-sync}} = \lceil n/2 \rceil - 1$ optimum-steps.*

In Fig. 3, we illustrate a space-time diagram for synchronizing a cellular space with the recursive-halving marking. Note that the general G is on the center cell(s) at time $t = 0$.

4 An Optimum-Time 2D FSSP Algorithm

4.1 Overview of the Algorithm

We assume that an initial general G is on the north-west corner cell C_{11} of a given rectangular array of size $m \times n$. The algorithm consists of four phases: decomposing the array into four subarrays, a marking phase for these subarrays, finding a center point of the given array, and a final synchronization phase. An overview of the 2D synchronization algorithm $\mathcal{A}$ is as follows:

Step 1. Decompose a given array into four subarrays. They consists of left and right triangles and upper and lower trapezoids, each denoted by α_0, α_1 and β_0, β_1, respectively, in the case $m \leq n$. See Fig. 4 (i). The case $m > n$ can be treated in a similar way.

Step 2. Start the recursive-halving marking for cells on each column in α_0 and α_1 and for each row in β_0 and β_1. See Fig. 4 (ii) and (iii).

Step 3. Find a *center cell(s)* of the given array and **generate** a new general(s) on the center cell(s). See Fig. 4 (iv).

Step 4. Synchronize each column in α_0 and α_1 and each row in β_0 and β_1 using Lemma 4, initiated by the general generated in Step 3. See Fig. 4 (v). This yields the final synchronization of the array.

4.2 Algorithm $\mathcal{A}$

We assume that $m = 2m_1 + 1, n = 2n_1 + 1$, where $m_1, n_1 \geq 1$. Other cases such as case 1: $m = 2m_1 + 1, n = 2n_1$, case 2: $m = 2m_1, n = 2n_1 + 1$, and case 3: $m = 2m_1, n = 2n_1$ can be treated in a similar way and the algorithm given below operates in optimum-steps. Operations in those four phases given above are as follows:

- **Decomposition:** The array is decomposed into four subarrays by four signals s_{D_i}, $1 \leq i \leq 4$, shown in Fig. 4(ii). These signals are generated by corner cells $C_{1,1}$, $C_{m,1}$, $C_{1,n}$, and $C_{m,n}$ at time $t = 0, m-1, n-1$ and $m+n-2$, respectively. Each signal travels along the diagonal and prints a special mark acting as a delimiter for the decomposition.

  First, the general G on $C_{1,1}$ generates three signals s_{H_1}, s_{V_1}, and s_{D_1} at time $t = 0$. Their operations are as follows:

  - **Signal s_{H_1}:** The s_{H_1}-signal travels along the 1st row at 1/1-speed and reaches $C_{1,n}$ at time $t = n - 1$. Then, it splits into two signals, s_{V_3} and s_{D_3}. The signal s_{V_3} travels downwards along the nth column at 1/1 speed

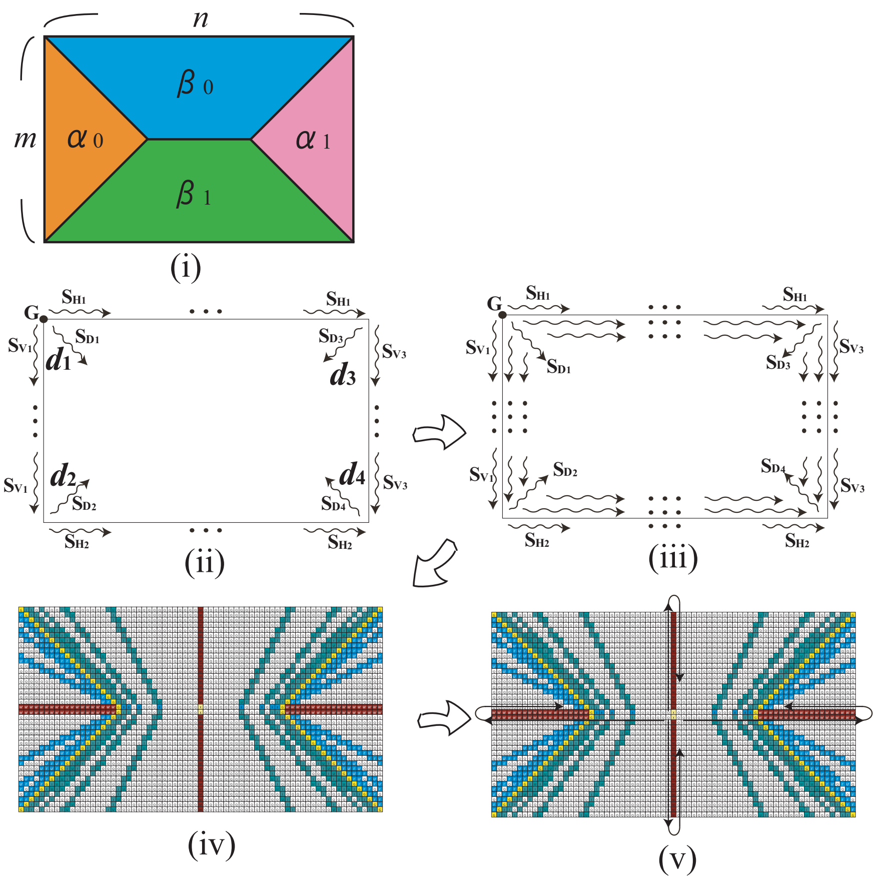


Fig. 4. Decomposition of a given rectangular array into four subarrays (i), signal propagations for the decomposition, marking and final synchronization (ii-v)

and reaches $C_{m,n}$ at time $t = m + n - 2$. It generates the signal $\mathbf{s}_{D_4}$. The signal s_{D_3} travels along the diagonal line at 1/1-speed and prints the delimiter.

- **Signal $\mathbf{s}_{V_1}$**: The s_{V_1}-signal travels along the 1st column at 1/1-speed and reaches $C_{m,1}$ at time $t = m - 1$. Then, it splits into two signals, $\mathbf{s}_{H_2}$ and $\mathbf{s}_{D_2}$. The signal s_{H_2} travels along the mth row at 1/1-speed and reaches $C_{m,n}$ at time $t = m + n - 2$. The signals s_{H_2} and s_{V_3} reach $C_{m,n}$ at time $t = m + n - 2$ simultaneously and generate the signal $\mathbf{s}_{D_4}$. The signals s_{D_2} and s_{D_4} travel along the diagonal at 1/1-speed and prints the delimiter.

- **Signal s_{D_1}**: The s_{D_1}-signal travels along the diagonal at 1/2-speed. The slow speed 1/2 is important for the recursive-halving marking in order to find end delimiter in each column and row.

For any i such that $1 \le i \le 4$, let d_i denote the delimiter marked by the diagonal signal s_{D_i}. We denote each row in β_0 delimited by d_1 and d_3 as $\mathrm{r}_{\beta_0 i}$ for $1 \le i \le \lceil m/2 \rceil$. Each row in β_1 delimited by d_2 and d_4 is denoted as $\mathrm{r}_{\beta_1 i}$ for $1 \le i \le \lceil m/2 \rceil$. The index i is counted from outside in β_0 and β_1. The ith column in α_0 and α_1 is also denoted by $\mathrm{c}_{\alpha_0 i}$ and $\mathrm{c}_{\alpha_1 i}$ for each i such that $1 \le i \le \lceil m/2 \rceil$, respectively. The index counting is also made from outside.

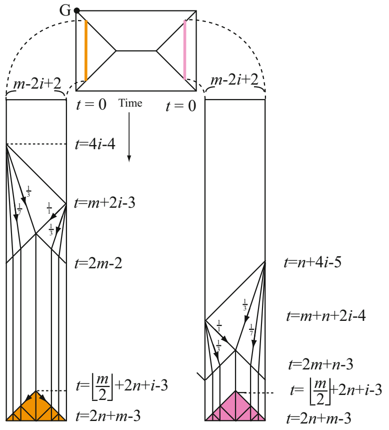


Fig. 5. Space-time diagram for the column synchronization on the ith columns $\mathrm{c}_{\alpha_0 i}$, $\mathrm{c}_{\alpha_1 i}$ in α_0 and α_1, respectively.

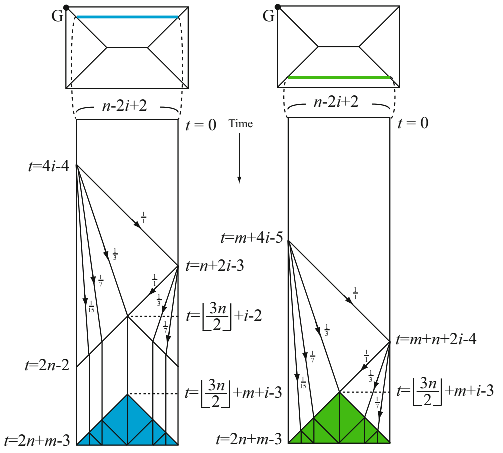


Fig. 6. Space-time diagram for the row synchronization on the ith rows $\mathrm{r}_{\beta_0 i}$, $\mathrm{r}_{\beta_1 i}$ in β_0 and β_1, respectively.

- **Recursive-Halving Marking:** Once the diagonal signal s_{D_1} prints the delimiter d_1 on a diagonal cell, it starts the recursive-halving marking operation for each column in α_0 and for each row in β_0. The cell with the delimiter d_1 is the starting point of the recursive-halving marking for each column and row. The end point for the column and row recursive-halving marking is the delimiter d_2 and d_3, respectively. See Fig. 4 (iii). The delimiter d_2 and d_3 act as not only an ending point for the marking but also a starting point for the next marking. The delimiter d_4 acts as an end point for the marking.

  As for the column marking in α_0 and α_1, the cell $\mathrm{C}_{i,i}$ starts the marking operation for the ith column in α_0 of length $m-2i+2$ at time $t = 4i-4$. The cell $\mathrm{C}_{i,n-i}$ starts the marking operation for the ith column in α_1 of length $m-2i+2$ at time $t = n+4i-5$. See Fig. 5. As for the row marking in β_0 and β_1, the cell $\mathrm{C}_{i,i}$ starts the marking operation for the ith row in β_0 of length $n-2i+2$ at time $t = 4i-4$. The cell $\mathrm{C}_{m-i,i}$ starts the marking operation for the ith row in β_1 of length $n-2i+2$ at time $t = m+4i-5$. See Fig. 6.

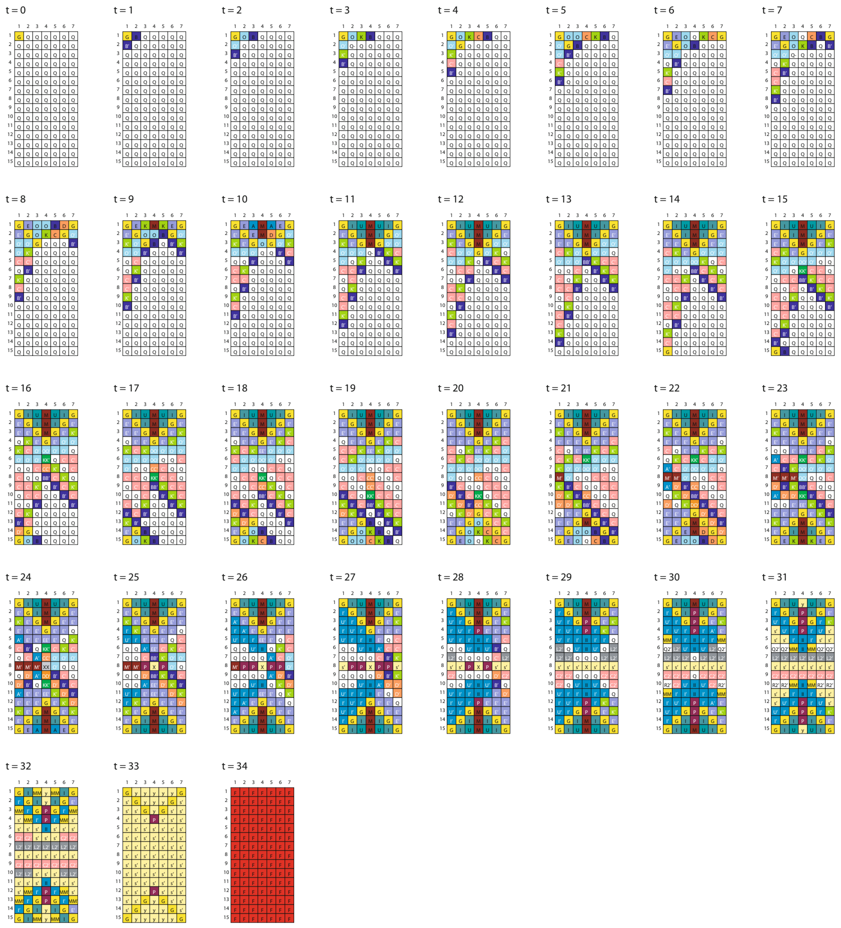


Fig. 7. Snapshots of the synchronization algorithm $\mathcal{A}$ on a 15×7 array

- **Finding a Center Point of the Array:** The center cell of the 1st column of length m in α_0 can be identified and marked at time $t = 3m_1$. The center marking in the column is propagated in the right direction at speed 1/1 on the row where it is generated, additionally requiring n_1 steps. The center cell of the 1st row of length n in β_0 can be identified and marked at time $t = 3n_1$. The center marking in the 1st row is propagated downward at speed 1/1 on the column where it is generated, additionally requiring m_1 steps. The center cell of the array is identified and marked at time $t_{\text{2D-center}} = \max(3m_1 + n_1, m_1 + 3n_1) = m_1 + n_1 - 1 + \max(2m_1 + 1, 2n_1 + 1) = m_1 + n_1 - 1 + \max(m, n)$.
- **Final Synchronization:** The general generated at the center of the array at time $t = t_{\text{2D-center}}$ acts as a general for the final synchronization. First, it send out a signal s_{sync} into four directions: leftward, rightward, upward, and downward. See Fig. 4 (v). It propagates at 1/1 speed and arrives at center

cells on the 1st column in α_0 and α_1 at time $t = t_{2\mathrm{D-center}} + n_1$. It also reaches at center cells on the 1st row in β_0 and β_1 at time $t = t_{2\mathrm{D-center}} + m_1$, respectively. Then, the signal reflects in the reverse direction and initiates the final synchronization for each row and column by propagating at 1/1 speed. For any i such that $1 \le i \le \lceil m/2 \rceil$, the ith column in α_0 and α_1 can be synchronized at time $t_{2\mathrm{D-sync}} = t_{2\mathrm{D-center}} + n_1 + i - 1 + \lceil (m - 2i + 2)/2 \rceil - 1 = 2m_1 + 1 + 2n_1 + 1 + \max(m, n) - 3 = m + n + \max(m, n) - 3$. In a similar way, For any i such that $1 \le i \le \lceil m/2 \rceil$, the ith row in β_0 and β_1 can be synchronized at time $t_{2\mathrm{D-sync}} = t_{2\mathrm{D-center}} + m_1 + i - 1 + \lceil (n - 2i + 2)/2 \rceil - 1 = 2m_1 + 1 + 2n_1 + 1 + \max(m, n) - 3 = m + n + \max(m, n) - 3$.
Thus all columns in α_0 and α_1 and all rows in β_0 and β_1 can be synchronized simultaneously at time $t = m + n + \max(m, n) - 3$.

Figures 5 and 6 show the space-time diagram for the column synchronization on the ith columns $\mathrm{c}_{\alpha_0 i}$, $\mathrm{c}_{\alpha_1 i}$ in α_0 and α_1 and for the row synchronization on the ith rows $\mathrm{r}_{\beta_0 i}$, $\mathrm{r}_{\beta_1 i}$ in β_0 and β_1, respectively.

One can see that each marking operation has been finished before the arrival of the first synchronization signal. The algorithm given above operates in optimum-steps in a similar way for the rectangles such as the case 1: $m = 2m_1 + 1, n = 2n_1$, case 2: $m = 2m_1, n = 2n_1 + 1$, and case 3: $m = 2m_1, n = 2n_1$.

Thus we have:

Theorem 5. The synchronization algorithm $\mathcal{A}$ can synchronize any $m \times n$ rectangular array in optimum $m + n + \max(m, n) - 3$ steps.

We have implemented the algorithm $\mathcal{A}$ on a 2D cellular automaton having 63 states and 25317 local rules, which has been verified for successful synchronization on any arrays of size $m \times n$ such that $2 \le m, n \le 200$. In Figures 7 and 8 we present some snapshots of the synchronization processes of the algorithm $\mathcal{A}$ on 15×7 and 7×14 arrays, respectively.

4.3 Generalized Algorithm

Now we are going to consider an extension of the optimum-time FSSP algorithm $\mathcal{A}$ to the generalized case where the general can be at any position of the array. We assume that an initial general G is on the cell $\mathrm{C}_{r,s}$ of a given array of size $m \times n$, where $1 \le r \le m, 1 \le s \le n$. As for the lower-bound of synchronization steps in 2D generalized FSSP algorithms, Umeo, Nishide, and Kubo [2012] has given it in the following theorem:

Theorem 6$^{\text{Umeo et. al [2012]}}$. There exists no 2D cellular automaton that can synchronize any 2D array of size $m \times n$ with an initial general on $\mathrm{C}_{r,s}$ in less than $m + n + \max(m, n) - \min(r, m - r + 1) - \min(s, n - s + 1) - 1$ steps, where $1 \le r \le m, 1 \le s \le n$.

The generalized 2D FSSP algorithm expanded operates in a similar way as in the case of the general on the north-west corner, and it consists of four operating phases: decomposing the array into four subarrays, a marking phase for these

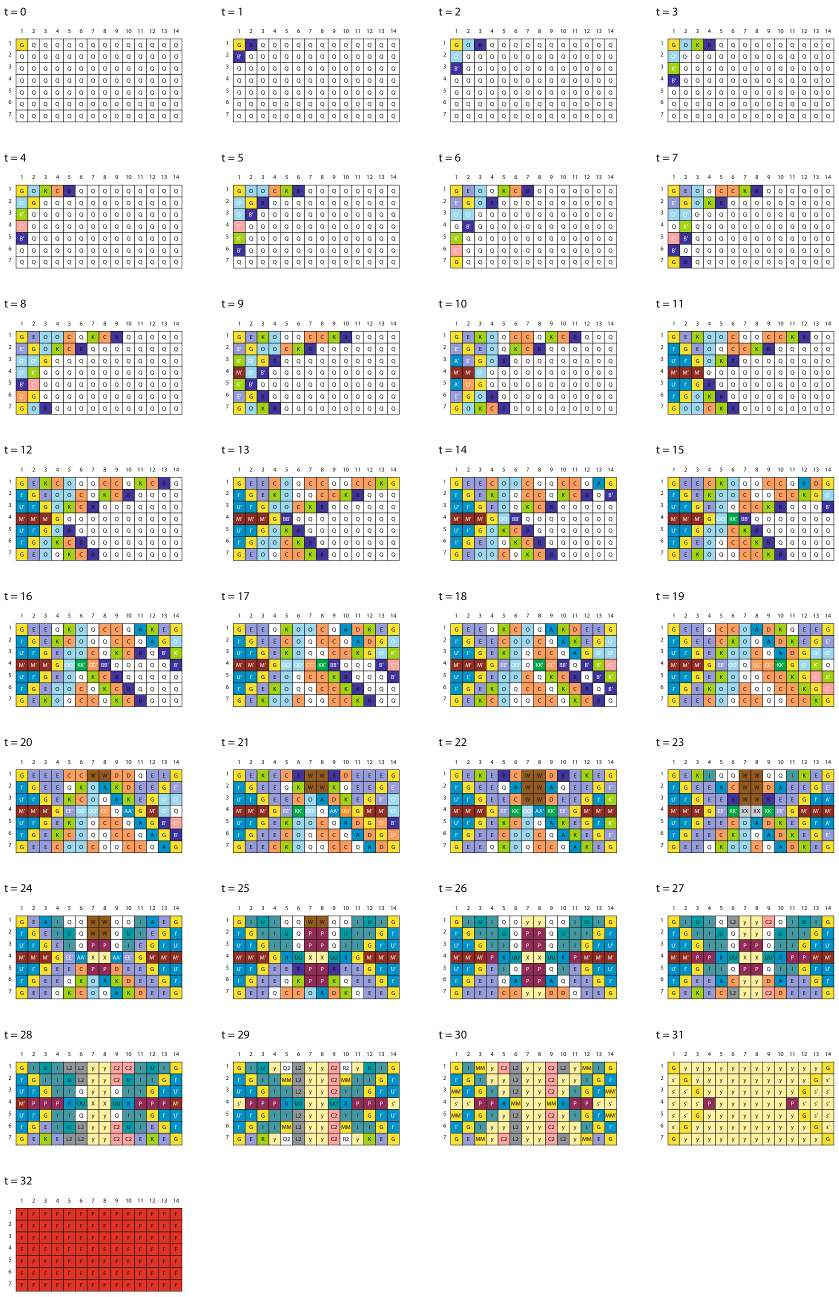


Fig. 8. Snapshots of the synchronization algorithm $\mathcal{A}$ on a 7×14 array

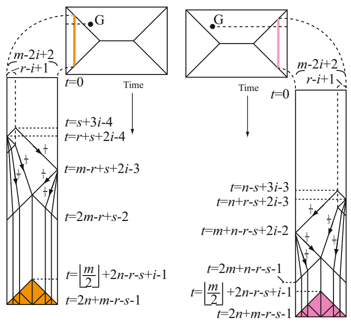


Fig. 9. Space-time diagram for the column synchronization on the ith columns in α_0 and α_1, respectively

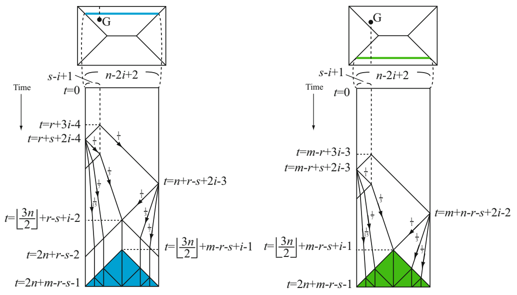


Fig. 10. Space-time diagram for the row synchronization on the ith rows in β_0 and β_1, respectively

subarrays, finding a center cell(s) of the given array, and a final synchronization phase. A decomposition the array is made in a similar way in the case where the initial general is on $C_{1,1}$. By starting the recursive-halving marking from on $C_{r,s}$, we can shorten the synchronization steps by $\min(r, m-r+1)+\min(s, n-s+1)-2$. Due to the space available, we omit the details of the algorithm. Figures 9 and 10 show the space-time diagram for the column synchronization on the ith columns in α_0 and α_1 and for the row synchronization on the ith rows in β_0 and β_1, respectively. The generalized 2D optimum-time FSSP algorithm is stated as follows:

Theorem 7. There exists an optimum-time synchronization algorithm that can synchronize any $m \times n$ rectangular array with a general at $C_{r,s}$ in optimum $m + n + \max(m, n) - \min(r, m - r + 1) - \min(s, n - s + 1) - 1$ steps, where $1 \le r \le m, 1 \le s \le n$.

5 Conclusions

We have proposed a new recursive-halving based optimum-time synchronization algorithm that can synchronize any rectangle 2D arrays of size $m \times n$ with a general at one corner in $m + n + \max(m, n) - 3$ steps. The algorithm is quite different from previous designs and it can be easily generalized to 2D arrays with a general at any position of the array. The algorithm is isotropic concerning the side-lengths of 2D arrays and its correctness is transparent and easily verified. A smaller-state realization on a 2D cellular automaton would be possible.

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, p. 144. MIT (1969)

3. Gerken, H.D.: Über Synchronisations Probleme bei Zellularautomaten. *Diplomarbeit*, Institut für Theoretische Informatik, Technische Universität Braunschweig, p. 50 (1987)
4. Goto, E.: A minimal time solution of the firing squad problem. Dittoed course notes for Applied Mathematics, vol. 298, pp. 52–59. Harvard University (1962)
5. Grasselli, A.: Synchronization of cellular arrays: The firing squad problem in two dimensions. Information and Control 28, 113–124 (1975)
6. Moore, E.F.: The firing squad synchronization problem. In: Moore, E.F. (ed.) Sequential Machines, Selected Papers, pp. 213–214. Addison-Wesley, Reading (1964)
7. Schmid, H.: Synchronisationsprobleme für zelluläre Automaten mit mehreren Generälen. Diplomarbeit, Universität Karsruhe (2003)
8. Shinahr, I.: Two- and three-dimensional firing squad synchronization problems. Information and Control 24, 163–180 (1974)
9. Szwerinski, H.: Time-optimum solution of the firing-squad-synchronization-problem for n-dimensional rectangles with the general at an arbitrary position. Theoretical Computer Science 19, 305–320 (1982)
10. 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 (2009)
11. Umeo, H., Hisaoka, M., Akiguchi, S.: A twelve-state optimum-time synchronization algorithm for two-dimensional rectangular cellular arrays. In: Calude, C.S., Dinneen, M.J., Păun, G., Jesús Pérez-Jímenez, M., Rozenberg, G. (eds.) UC 2005. LNCS, vol. 3699, pp. 214–223. Springer, Heidelberg (2005)
12. Umeo, H., Nishide, K., Kubo, K.: A simple optimum-time FSSP algorithm for multi-dimensional cellular automata. In: Proc. of Automata 2012 and JAC 2012. EPTCS, pp. 66–80 (2012)
13. Waksman, A.: An optimum solution to the firing squad synchronization problem. Information and Control 9, 66–78 (1966)

MathCloud: Publication and Reuse of Scientific Applications as RESTful Web Services

Alexander Afanasiev, Oleg Sukhoroslov, and Vladimir Voloshinov

Institute for Information Transmission Problems of the Russian Academy of Sciences,
Bolshoy Karetny per. 19, Moscow, 127994, Russia
oleg.sukhoroslov@gmail.com

Abstract. The paper presents MathCloud platform which enables wide-scale sharing, publication and reuse of scientific applications as RESTful web services. A unified interface of computational web service based on REST architectural style is proposed. Main components of MathCloud platform including service container, service catalogue, workflow management system, and security mechanism are described. In contrast to other similar efforts based on WS-* specifications, the described platform provides a more lightweight solution with native support for modern Web applications. The platform has been successfully used in several applications from various fields of computational science that confirm the viability of proposed approach and software platform.

Keywords: computational web service, service-oriented scientific environment, software as a service, REST, service container, service catalogue, workflow.

1 Introduction

Modern scientific research is closely related to complex computations and analysis of massive datasets. Computational Science is a rapidly growing field that uses advanced computing and data analysis to solve complex scientific and engineering problems. In their research scientists actively use software applications that implement computational algorithms, numerical methods and models of complex systems. Typically, these applications require massive amounts of calculations and are often executed on supercomputers or distributed computing systems.

The increasing complexity of problems being solved requires simultaneous use of several computational codes and computing resources. This leads to an increased complexity of applications and computing infrastructures. The multi- and interdisciplinary nature of modern science requires collaboration within distributed research projects including coordinated use of scientific expertise, software and resources of each partner. This brings a number of problems faced by a scientist in a day-to-day research.

The reuse of existing computational software is one of key factors influencing research productivity. However, the increased complexity of such software means that it often requires specific expertise in order to install, configure and run it that is beyond the expertise of an ordinary researcher. This specific expertise also involves configuration and use of high performance computing resources required to run the software. In some cases such expertise can be provided by IT support staff, but this brings additional operating expenses that can be prohibitive for small research teams. The problem amplifies in case of actively evolving software which means that it has to be upgraded or reinstalled on a regular basis. In addition to problem-specific parameters many applications require specification of additional runtime parameters such as number of parallel processes. Mastering these parameters also requires additional expertise that sometimes can only be provided by software authors.

Modern supercomputing centers and grid infrastructures provide researchers with access to high performance computing resources. Such facilities also provide access to preinstalled popular computational packages which partially solves the aforementioned problem. Researchers can also use such facilities to run arbitrary computational code. But here lies another problem. In this case, in addition to master the software, the researcher also has to master the subtleties of working with the command line and the batch system of supercomputer or grid middleware. About 30 years ago such interface was taken for granted, but in the eyes of a modern researcher it looks the same as a text web browser - awkward and archaic. Without radically changing their interface, scientific computing facilities have grown, become more complex inside and harder to use.

The third issue faced by a modern computational scientist is related to the need to combine multiple applications such as models or solvers in order to solve a complex problem. Typically, this issue represents a complex problem on its own which naturally includes all the issues discussed previously. It also brings an important problem of interoperability between computational applications written by different authors. Some applications are designed without interoperability in mind which means that this issue has to be resolved by researcher itself.

The described problems severely reduce the research productivity by not allowing scientists to focus on real problems to be solved. Therefore there is a huge demand for high-level interfaces and problem solving environments that hide the complexity of applications and infrastructure from a user.

The most promising approach for taming complexity and enabling reuse of applications is the use of service-oriented architecture (SOA). SOA consists of a set of principles and methodologies for provision of applications in the form of remotely accessible, interoperable services. The use of SOA can enable wide-scale sharing, publication and reuse of scientific applications, as well as automation of scientific tasks and composition of applications into new services [1].

The provision of applications as services is closely related to “Software as a Service” (SaaS) software delivery model implemented nowadays by many web and cloud computing services. This model has several advantages in comparison to traditional software delivery such as ability to run software without installation using a web browser, centralized maintenance and accelerated feature delivery. The ubiquity of SaaS applications and the ability to access these applications via programmable APIs have spawned development of mashups that combine data, presentation and functionality from multiple services, creating a composite service.

A key observation here is that, in essence, the aforementioned issues are not unique to scientific computing. However, it is still an open question how existing approaches, such as SOA, SaaS and Web 2.0, can be efficiently applied in the context of scientific computing environments.

The paper presents MathCloud platform which enables wide-scale sharing, publication and reuse of scientific applications as RESTful web services. Section 2 introduces a unified remote interface of computational web service based on REST architectural style. Section 3 describes main components of MathCloud platform including service container, service catalogue, workflow management system, and security mechanism. Section 4 presents applications and experimental evaluation of created platform. Section 5 discusses related work.

2 Unified Interface of Computational Web Service

Currently, the dominant technology for building service-oriented systems are Web services based on SOAP protocol, WSDL and numerous WS-* specifications (hereinafter referred to as “big Web services”). A common criticism of big Web services is their excessive complexity and incorrect use of core principles of the Web architecture [2]. The advantages of big Web services mostly apply to complex application integration scenarios and business processes that occur in enterprise systems, while rarely present in Web 2.0 applications that favor ease-of-use and ad hoc integration [3].

The most promising alternative approach to implementation of web services is based on the REST (Representational State Transfer) architectural style [4]. Thanks to the uniform interface for accessing resources, the use of core Web standards and the presence of numerous proven implementations, REST provides a lightweight and robust framework for development of web services and related client applications. This is confirmed by a proliferation of the so-called RESTful web services [2], especially within Web 2.0 applications.

Assuming that service-oriented scientific environments should also emphasize ease-of-use and ad hoc integration of services, we propose to implement computational services as RESTful web services with a unified interface [5]. This interface or REST API is based on the following abstract model of a computational service. A service processes incoming client's requests to solve specific problems. A client's request includes a parameterized description of the problem, which is represented as a set of input parameters. Having successfully processed the request, the service returns the result represented as a set of output parameters to the client.

The proposed unified interface of computational web service is formed by a set of resources identified by URIs and accessible via standard HTTP methods (Table. 1). The interface takes into account features of computational services by supporting asynchronous request processing and passing large data parameters. Also, in accordance with the service-oriented approach, the interface supports introspection, i.e., obtaining information about the service and its parameters.

Table 1. REST API of computational web service

Resource	GET	POST	DELETE
Service	Get service description	Submit new request (create job)	
Job	Get job status and results		Cancel job, delete job data
File	Get file data		

The service resource supports two HTTP methods. GET method returns the service description. POST method allows a client to submit a request to server. The request body contains values of input parameters. Some of these values may contain identifiers of file resources. In response to the request, the service creates a new subordinate job resource and returns to the client identifier and current representation of the job resource.

The job resource supports GET and DELETE methods. GET method returns job representation with information about current job status. If the job is completed successfully, then the job representation also contains job results in the form of values of output parameters. Some of these values may contain identifiers of file resources.

The DELETE method of job resource allows a client to cancel job execution or, if the job is already completed, delete job results. This method destroys the job resource and its subordinate file resources.

The file resource represents a part of client request or job result provided as a remote file. The file contents can be retrieved fully or partially via the GET method of HTTP or other data transfer protocol.

Note that the described interface doesn't prescribe specific templates for resource URIs which may vary between implementations. It is desirable to respect the described hierarchical relationships between resources while constructing these URIs.

The described interface supports job processing in both synchronous and asynchronous modes. Indeed, if the job result can be immediately returned to the client, then it is transmitted inside the returned job resource representation along with the indication of DONE state. If, however, the processing of request takes time, it is stated in the returned job resource representation by specifying the appropriate job state (WAITING or RUNNING). In this case, the client uses the obtained job resource identifier for further checking of job state and obtaining its results.

The proposed REST API is incomplete without considering resource representation formats and means of describing service parameters. The most widely used data representation formats for Web services are XML and JSON. Among these JSON has been chosen for the following reasons. First, JSON provides more compact and readable representation of data structures, while XML is focused on representation of arbitrary documents. Second, JSON supports native integration with JavaScript language simplifying creation of modern Ajax based Web applications.

A known disadvantage of JSON is the lack of standard tools for description and validation of JSON data structures comparable to XML Schema. However, there is an active ongoing work on such format called JSON Schema [6]. This format is used for description of input and output parameters of computational web services within the proposed REST API.

3 MathCloud Platform

MathCloud platform [7] is a software toolkit for building, deployment, discovery and composition of computational web services using the proposed REST API. This section presents main components of MathCloud platform.

3.1 Service Container

Service container codenamed Everest represents a core component of the platform. Its main purpose is to provide a high-level framework for development and deployment of computational web services. Everest simplifies service development by means of ready-to-use adapters for common types of applications. The container also implements a universal runtime environment for such services based on the proposed REST API. The architecture of Everest is presented in Fig. 1.

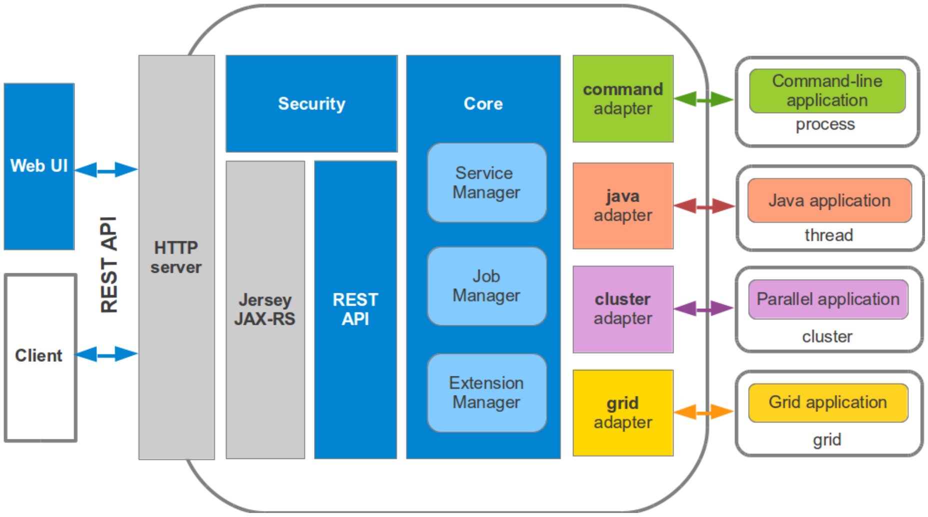


Fig. 1. Architecture of service container

The service container is based on Jersey library, a reference implementation of JAX-RS (Java API for RESTful Web Services) specification. The container uses built-in Jetty web server for interaction with service clients. Incoming HTTP requests are dispatched to Jersey and then to the container. Everest is processing client requests in accordance with configuration information.

The Service Manager component maintains a list of services deployed in the container and their configuration. This information is read at startup from configuration files. The configuration of each service consists of two parts:

- Public service description which is provided to service clients;
- Internal service configuration which is used during request processing.

The Job Manager component manages the processing of incoming requests. The requests are converted into asynchronous jobs and placed in a queue served by a configurable pool of handler threads. During job processing, handler thread invokes adapter specified in the service configuration.

The components that implement processing of service requests (jobs) are provided in the form of pluggable adapters. Each adapter implements a standard interface through which the container passes request parameters, monitors the job state and receives results.

Currently the following universal adapters are implemented.

The Command adapter converts service request to an execution of specified command in a separate process. The internal service configuration contains the command to execute and information about mappings between service parameters and command line arguments or external files.

The Java adapter performs invocation of a specified Java class inside the current Java virtual machine, passing request parameters inside the call. The specified class must implement standard Java interface. The internal service configuration includes the name of the corresponding class.

The Cluster adapter performs translation of service request into a batch job submitted to computing cluster via TORQUE resource manager. The internal service configuration contains the path to the batch job file and information about mappings between service parameters and job arguments or files.

The Grid adapter performs translation of service request into a grid job submitted to the European Grid Infrastructure, which is based on gLite middleware. This adapter can be used both to convert existing grid application to service and to port existing service implementation to the grid. The internal service configuration contains the name of grid virtual organization, the path to the grid job description file and information about mappings between service parameters and job arguments or files.

Note that the all adapters, except Java, support converting of existing applications to services by writing only a service configuration file, i.e., without writing a code. This feature makes it possible for unskilled users to publish as services a wide range of existing applications. Besides that, the support for pluggable adapters allows one to attach arbitrary service implementations and computing resources.

Each service deployed in Everest is published via the proposed REST API. In addition to this, container automatically generates a complementary web interface allowing users to access the service via a web browser.

3.2 Service Catalogue

The main purpose of service catalogue is to support discovery, monitoring and annotation of computational web services. It is implemented as a web application with interface and functionality similar to modern search engines.

After the service is deployed in the service container it can be published in the catalogue by providing a URI of the service and a few tags describing it. The catalogue retrieves service description via the unified REST API, performs indexing and stores description along with specified tags in a database.

The catalogue provides a search query interface with optional filters. It supports full text search in service descriptions and tags. Search results consist of short snippets of each found service with highlighted query terms and a link to full service description.

In order to provide current information on service availability the catalogue periodically pings published services. If a service is not available it is marked accordingly in search results. The catalogue also implements some experimental features similar to collaborative Web 2.0 sites, e.g., ability to tag services by users.

3.3 Workflow Management System

In order to simplify composition of services a workflow management system is implemented [8]. The system supports description, storage, publication and execution of workflows composed of multiple services. Workflows are represented as directed acyclic graphs and described by means of a visual editor. The described workflow can be published as a new composite service and then executed by sending request to this service. The system has client-server architecture. The client part of the system is represented by workflow editor, while the server part is represented by the workflow management service.

Fig. 2 shows the interface of the workflow editor. It is inspired by Yahoo! Pipes and implemented as a Web application in JavaScript language. This makes it possible to use the editor on any computer running a modern web browser.

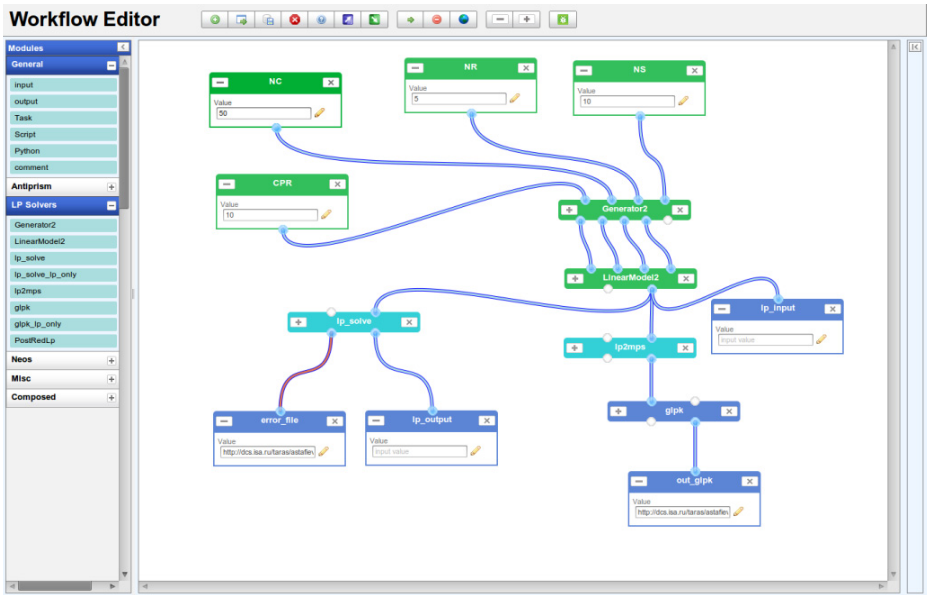


Fig. 2. Graphical workflow editor

The workflow is represented in the form of a directed acyclic graph whose vertices correspond to workflow blocks and edges define data flow between the blocks. Each block has a set of inputs and outputs displayed in the form of ports at the top and at the bottom of the block respectively. Each block implements a certain logic of processing of input data and generating of output data. Data transfer between blocks is realized by connecting the output of one block to the input of another block. Each input or output has associated data type. The compatibility of data types is checked during connecting the ports.

The introduction of a service in a workflow is implemented by creating a new Service block and specifying the service URI. It is assumed that the service implements the unified REST API. This allows the editor to dynamically retrieve service description and extract information about the number, types and names of input and output parameters of the service. This information is used to automatically generate the corresponding input and output ports of the block.

The unified REST API provides a basis for service interoperability on the interface level. The user can connect any output of one service with any input of another service if both ports have compatible data types. However, it is important to note that the system doesn't check the compatibility of data formats and semantics of the corresponding parameters. It is the task of the user to ensure this.

An important feature of the editor is ability to run a workflow and display its state during the execution. Before the workflow can be run it is necessary to set the values of all input parameters of the workflow via the appropriate Input blocks. After the user clicks on the Run button, the editor makes a call with the specified input parameters to the composite service representing the workflow. Then the editor performs a periodic check of the status of running job, which includes information about states of individual blocks of the workflow. This information is displayed to the user by painting each workflow block in the color corresponding to its current state. After successful completion of the workflow, the values of workflow output parameters are displayed in the Output blocks. Each workflow instance has a unique URI which can be used to open the current state of the instance in the editor at any time. This feature is especially useful for long-running workflows.

The workflow management service (WMS) performs storage, deployment and execution of workflows created with the described editor. In accordance with the service-oriented approach the WMS deploys each saved workflow as a new service. The subsequent workflow execution is performed by sending request to the new composite service through the unified REST API. Such requests are processed by the workflow runtime embedded in WMS. The WMS is implemented as a RESTful web service. This provides a most convenient way to interact with the WMS from the workflow editor.

3.4 Security

All platform components use common security mechanism (Fig. 3) for protecting access to services. It supports authentication, authorization and a limited form of delegation based on common security technologies.

Authentication of services is implemented by means of SSL server certificates. Authentication of clients is implemented via two mechanisms. The first one is standard X.509 client certificate. The second is Loginza service which supports authentication via popular identity providers (Google, Facebook, etc.) or any OpenID provider. The latter mechanism, which is available only for browser clients, is convenient for users who don't have a certificate.

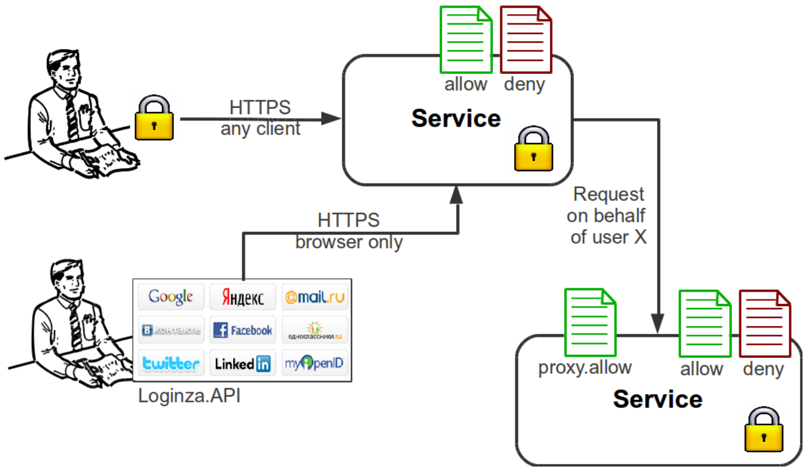


Fig. 3. Security mechanism

Authorization is supported by means of allow and deny lists which enable service administrator to specify users which should or should not have access to a service. A user can be specified using its certificate's distinguished name or OpenId identifier.

A well-known security challenge in service-oriented environments is providing a mechanism for a service to act on behalf of a user, i.e., invoke other services. A common use case is a workflow service which needs to access services involved in the workflow on behalf of a user invoked the service. For such cases a proxying mechanism is implemented by means of a proxy list which enable service administrator to specify certificates of services that are trusted to invoke the service on behalf of users. In comparison to proxy certificate mechanism used in grids, this approach is more limited but provides a lightweight solution compatible with the proposed REST API.

3.5 Clients

The platform provides Java, Python and command-line clients for accessing services from external applications and implementing complex workflows. Since the access to services is implemented via REST API, one can use any standard HTTP library or client (e.g., curl) to interact with services. Also, thanks to using JSON format, services can be easily accessed from JavaScript applications via Ajax calls. This simplifies development of modern Web-based interfaces to services, in contrast to approaches based on big Web services and XML data format.

4 Applications

MathCloud platform has been used in several applications from various fields of computational science. This section describes some of these applications and summarizes general conclusions drawn from their development.

One of the first applications of MathCloud platform concerns an “error-free” inversion of ill-conditioned matrix [9], a well-known challenging task in computational science. The application uses symbolic computation techniques available in computer algebra systems (CAS) which require substantial computing time and memory. To address this issue a distributed algorithm of matrix inversion has been implemented via Maxima CAS system exposed as a computational web service. The algorithm was implemented as a workflow based on block decomposition of input matrix and Schur complement. The approach has been validated by inversion of Hilbert matrices up to 500×500.

The matrix inversion application provided an opportunity to evaluate performance of all platform components, in particular with respect to passing large amounts of data between services. In the case of extremely ill-conditioned matrices the symbolic representation of final and intermediate results reached up to hundreds of megabytes. Table 2 presents obtained performance results including serial execution time in Maxima, parallel execution time in MathCloud (using 4-block decomposition) and observed speedup. Additional analysis revealed that the overhead introduced by the platform including data transfer is about 2-5% of total computing time.

Table 2. Performance of Hilbert (NxN) matrix inversion application in MathCloud

N	Serial execution time in Maxima, minutes	Parallel execution time in MathCloud, minutes	Speedup
250	8	5	1,60
300	15	8	1,88
350	27	13	2,08
400	45	20	2,25
450	72	30	2,40
500	109	40	2,73

Another MathCloud application has been developed for interpreting the data of X-ray diffractometry of carbonaceous films by means of solving optimization problems within a broad class of carbon nanostructures [10]. The application is implemented as a workflow which combines parallel calculations of scattering curves for individual nanostructures (performed by a grid application) with subsequent solution of optimization problems (performed by three different solvers running on a cluster) to determine the most probable topological and size distribution of nanostructures. All these parts of computing scheme (and a number of additional steps, e.g., data preparation, post-optimal processing and plotting) have been implemented as computational web services. The application helped to reveal the prevalence of low-aspect-ratio toroids in tested films [11].

A recent work [12-13] concerns a uniform approach to creation of computational web services related to optimization modeling. MathCloud platform is used within this work to integrate various optimization solvers intended for basic classes of mathematical programming problems and translators of AMPL optimization modeling language. A number of created computational web services and workflows cover all basic phases of optimization modeling techniques: input of optimization problems' data, interaction with solvers, processing of solutions found.

A special service has been developed that implements dispatching of optimization tasks to a pool of solver services directly via AMPL translator's execution. These features enable running any optimization algorithm written as an AMPL script in distributed mode when all problems (and/or intermediate subproblems) are solved by remote optimization services. Independent problems are solved in parallel thus increasing overall performance in accordance with the number of available services. The proposed approach has been validated by the example of Dantzig–Wolfe decomposition algorithm for multi-commodity transportation problem.

The experience gained from application development shows that MathCloud platform can be efficiently applied for solving a wide class of problems. This class can be described as problems that allow decomposition into several coarse-grained suproblems (dependent or independent) that can be solved by existing applications represented as services. It is important to note that while MathCloud can be used as a parallel computing platform in homogeneous environments such as a cluster, it is generally not as efficient in this setting as dedicated technologies such as MPI. The main benefits of MathCloud are revealed in heterogeneous distributed environments involving multiple resources and applications belonging to different users and organizations.

The exposing of computational applications as web services is rather straightforward with MathCloud. From our experience it usually takes from tens of minutes to a couple of hours to produce a new service including service deployment and debugging. This is mainly due to the fact that the service interface is fixed and the service container provides a framework that implements all problem-independent parts of a service. That means that a user doesn't need to develop a service from scratch as it happens when using general purpose service-oriented platforms. In many cases service development reduces to writing a service configuration file. In other cases a development of additional application wrapper is needed which is usually accomplished by writing a simple shell or Python script.

The workflow development is somewhat harder, especially in the case of complex workflows. However, the workflow editor provides some means for dealing with this. First of all, it enables dividing complex workflow into several simpler sub-workflows by supporting publishing and composing of workflows as services. Second, it is possible to add custom workflow actions written in JavaScript or Python, for example to create complex string inputs for services from user data or to get additional timing. Finally, besides the graphical editor it is possible to download workflow in JSON format, edit it manually and upload back to WMS. These and other features provide rather good usability for practical use of MathCloud platform.

5 Related Work

The use of service-oriented approach in the context of scientific computing was proposed in [1]. Service-Oriented Science as introduced by Foster refers to scientific research enabled by distributed networks of interoperating services.

The first attempts to provide a software platform for Service-Oriented Science were made in Globus Toolkit 3 and 4 based on the Open Grid Services Architecture (OGSA) [14]. OGSA describes a service-oriented grid computing environment based on big Web services. Globus Toolkit 3/4 provided service containers for deployment of stateful grid services that extended big Web services. These extensions were documented in the Web Services Resource Framework (WSRF) specification [15]. It largely failed due to inherent complexity and inefficiencies of both specification and its implementations. Globus Toolkit 4 had steep learning curve and provided no tools for rapid deployment of existing applications as services and connecting services to grid resources.

There have been several efforts aiming at simplifying transformation of scientific applications into remotely accessible services. The Java CoG Kit [16] provided a way to expose legacy applications as Web services. It uses a serviceMap document to generate source code and WSDL descriptor for the Web service implementation. Generic Factory Service (GFac) [17] provides automatic service generation using an XML-based application description language. Instead of source code generation, it uses an XSUL Message Processor to intercept the SOAP calls and route it to a generic class that invokes the scientific application. SoapLab [18] is another toolkit that uses an application description language called ACD to provide automatic Web service wrappers.

Grid Execution Management for Legacy Code Architecture (GEMLCA) [19] implements a general architecture for deploying legacy applications as grid services. It implements an application repository and a set of WSRF-based grid services for deployment, execution and administration of applications. Instead of generation of different WSDLs for every deployed application as in GFac and SoapLab, GEMLCA uses generic interface and client for application execution. The execution of applications is implemented by submission of grid jobs through back-end plugins supporting Globus Toolkit and gLite middleware. GEMLCA was integrated with the P-GRADE grid portal [20] in order to provide user-friendly Web interfaces for application deployment and execution. The workflow editor of the P-GRADE portal supports connection of GEMLCA services into workflows using a Web-based graphical environment.

Opal [21] and Opal2 [22] toolkits provide a mechanism to deploy scientific applications as Web services with standard WSDL interface. This interface provides operations for job launch (which accepts command-line arguments and input files as its parameters), querying status, and retrieving outputs. In comparison to GEMLCA, Opal toolkit deploys a new Web service for each wrapped application. Opal also provides an optional XML-based specification for command-line arguments, which is used to generate automatic Web forms for service invocation. In addition to Base64 encoded inputs, Opal2 supports transfer of input files from remote URLs and via MIME attachments which greatly improves the performance of input staging.

It supports several computational back-ends including GRAM, DRMAA, Torque, Condor and CSF meta-scheduler.

The described toolkits have many similarities with the presented software platform, e.g., declarative application description, uniform service interface, asynchronous job processing. A key difference is related to the way services are implemented. While all mentioned toolkits use big Web services and XML data format, the MathCloud platform exposes applications as RESTful web services using JSON format. The major advantages of this approach are decreased complexity, use of core Web standards, wide adoption and native support for modern Web applications as discussed in Section 2.

The idea of using RESTful web services and Web 2.0 technologies as a lightweight alternative to big Web services for building service-oriented scientific environments was introduced in [23]. Given the level of adoption of Web 2.0 technologies relative to grid technologies, Fox et al. suggested the replacement of many grid components with their Web 2.0 equivalents. Nevertheless, to the authors' knowledge, there are no other efforts to create a general purpose service-oriented toolkit for scientific applications based on RESTful web services.

There are many examples of applying Web 2.0 technologies in scientific research in the form of Web-based scientific gateways and collaborative environments [20, 24-25]. While such systems support convenient access to scientific applications via Web interfaces, they don't expose applications as services thus limiting application reuse and composition.

There are many scientific workflow systems, e.g. [26-28]. The system described in the paper stands out among these by providing a Web-based interface, automatic publication of workflows as composite services and native support for RESTful web services.

6 Conclusion

The paper presented MathCloud platform which enables wide-scale sharing, publication and reuse of scientific applications as RESTful web services based on the proposed unified REST API. In contrast to other similar efforts based on Web Services specifications, it provides a more lightweight solution with native support for modern Web applications. MathCloud includes all core tools for building a service-oriented environment such as service container, service catalogue and workflow system. The platform has been successfully used in several applications from various fields of computational science that confirm the viability of proposed approach and software platform.

The future work will be focused on building a hosted Platform-as-a-Service (PaaS) for development, sharing and integration of computational web services based on the described software platform.

Acknowledgements. The work is supported by the Presidium of the Russian Academy of Sciences (the basic research program No.14) and the Russian Foundation for Basic Research (grant No. 11-07-00543-a).

References

1. Foster, I.: Service-Oriented Science. Science 308(5723), 814–817 (2005)
2. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media (2007)
3. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. "big" web services: making the right architectural decision. In: 17th International Conference on World Wide Web (WWW 2008), pp. 805–814. ACM, New York (2008)
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. dissertation, University of California, Irvine, Irvine, California (2000)
5. Sukhoroslov, O.V.: Unified Interface for Accessing Algorithmic Services in Web. Proceedings of ISA RAS 46, 60–82 (2009) (in Russian)
6. JSON Schema, `http://www.json-schema.org/`
7. MathCloud Project, `http://mathcloud.org/`
8. Lazarev, I.V., Sukhoroslov, O.V.: Implementation of Distributed Computing Workflows in MathCloud Environment. Proceedings of ISA RAS 46, 6–23 (2009) (in Russian)
9. Voloshinov, V.V., Smirnov, S.A.: Error-Free Inversion of Ill-Conditioned Matrices in Distributed Computing System of RESTful Services of Computer Algebra. In: 4th Intern. Conf. Distributed Computing and Grid-Technologies in Science and Education, pp. 257–263. JINR, Dubna (2010)
10. Neverov, V.S., Kukushkin, A.B., Marusov, N.L., et al.: Numerical Modeling of Interference Effects of X-Ray Scattering by Carbon Nanostructures in the Deposited Films from TOKAMAK T-10. Problems of Atomic Science and Technology. Thermonuclear Fusion, vol. 1, pp. 13–24 (2011) (in Russian)
11. Kukushkin, A.B., Neverov, V.S., Marusov, N.L., et al.: Few-nanometer-wide carbon toroids in the hydrocarbon films deposited in tokamak T-10. Chemical Physics Letters 506, 265–268 (2011)
12. Voloshinov, V.V., Smirnov, S.A.: On development of distributed optimization modelling systems in the REST architectural style. In: 5th Intern. Conf. Distributed Computing and Grid-Technologies in Science and Education. JINR, Dubna (2012)
13. Voloshinov, V.V., Smirnov, S.A.: Software Integration in Scientific Computing. Information Technologies and Computing Systems (3), 66–71 (2012) (in Russian)
14. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: Grid Services for Distributed System Integration. Computer 35(6), 37–46 (2002)
15. WS-Resource Framework, `http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf`
16. von Laszewski, G., Gawor, J., Krishnan, S., Jackson, K.: Commodity Grid Kits - Middleware for Building Grid Computing Environments. In: Grid Computing: Making the Global Infrastructure a Reality, ch. 25. Wiley (2003)
17. Kandaswamy, G., Fang, L., Huang, Y., Shirasuna, S., Marru, S., Gannon, D.: Building Web Services for Scientific Grid Applications. IBM Journal of Research and Development 50(2.3), 249–260 (2006)
18. SoapLab Web Services, `http://www.ebi.ac.uk/soaplab/`
19. Delaitre, T., Kiss, T., Goyeneche, A., Terstyanszky, G., Winter, S., Kacsuk, P.: GEMLCA: Running Legacy Code Applications as Grid Services. Journal of Grid Computing 3(1-2), 75–90 (2005)
20. Kacsuk, P., Sipos, G.: Multi-Grid, Multi-User Workflows in the P-GRADE Portal. Journal of Grid Computing 3(3-4), 221–238 (2005)

21. Krishnan, S., Stearn, B., Bhatia, K., Baldridge, K.K., Li, W., Arzberger, P.: Opal: Simple Web Services Wrappers for Scientific Applications. In: IEEE Intl. Conf. on Web Services (ICWS) (2006)
22. Krishnan, S., Clementi, L., Ren, J., Papadopoulos, P., Li, W.: Design and Evaluation of Opal2: A Toolkit for Scientific Software as a Service. In: 2009 IEEE Congress on Services (SERVICES-1 2009), pp. 709–716 (2009)
23. Fox, G., Pierce, M.: Grids Challenged by a Web 2.0 and Multicore Sandwich. Concurrency and Computation: Practice and Experience 21(3), 265–280 (2009)
24. McLennan, M., Kennell, R.: HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering. Computing in Science and Engineering 12(2), 48–52 (2010)
25. Afgan, E., Goecks, J., Baker, D., Coraor, N., Nekrutenko, A., Taylor, J.: Galaxy - a Gateway to Tools in e-Science. In: Yang, K. (ed.) Guide to e-Science: Next Generation Scientific Research and Discovery, pp. 145–177. Springer (2011)
26. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludäscher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), pp. 423–424 (2004)
27. Deelman, E., Singh, G., Su, M.H., et al.: Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Scientific Programming 13(3), 219–237 (2005)
28. Oinn, T., Greenwood, M., Addis, M., et al.: Taverna: lessons in creating a workflow environment for the life sciences. Concurrency Computation Practice and Experience 18(10), 1067–1100 (2006)

Enhanced Differential Evolution Entirely Parallel Method for Biomedical Applications

Konstantin Kozlov[1,⋆], Nikita Ivanisenko[2], Vladimir Ivanisenko[2], Nikolay Kolchanov[2], Maria Samsonova[1], and Alexander M. Samsonov[3]

[1] Dept. of Computational Biology, State Polytechnical University, St. Petersburg, 195251 Russia
{kozlov,samson}@spbcas.ru

[2] Institute of Cytology and Genetics, SB RAS, Novosibirsk, 630090 Russia
{ivanisenko,salix,kol}@bionet.nsc.ru

[3] The A.F. Ioffe Physical Technical Institute of the Russian Academy of Sciences, St. Petersburg, 194021 Russia
samsonov@math.ioffe.ru

Abstract. A considerable enhancement is proposed for the Differential Evolution Entirely Parallel (DEEP) method developed recently. A new selection rule was implemented in order to increase the robustness of DEEP. To simplify the approach a population is not divided now into branches, instead of it, several oldest individuals are substituted with the same number of the best ones after the predefined number of iterations. The individuals are selected on the basis of the number of generations, in which they survived without any change. We demonstrate how the enhanced DEEP provides new solutions to problems with several objective functions.

Keywords: differential evolution, optimization, biomedical applications.

1 Introduction

Models for biomedical applications usually are based on determination of unknown constants of underlying biochemical reactions, as well as parameters and coefficients. These unknowns are to be found as a solution to an inverse problem of mathematical modeling, i.e. by fitting model equations to data.

Development of reliable and easy-to-use algorithms and programs for solution of the inverse problem remains a challenging task due to diversity and high computational complexity of biomedical applications, as well as the necessity to treat large sets of heterogeneous data.

We propose a new modification of Parallel Differential Evolution approach, as an enhancement for the Differential Evolution Entirely Parallel (DEEP) method developed recently [1,2]. To increase the robustness of the procedure we implemented a new selection rule for Differential Evolution, in which several different objective functions are considered in order to accept or reject an offspring to

⋆ Corresponding author.

new generation. For computational complexity reduction the population is not to be divided into branches, but instead several oldest individuals are substituted with the same number of the best ones after predefined number of iterations. The individuals are selected on the basis of the number of generations, in which they survived without any change.

The DEEP method was successfully applied already to several problems, e.g., to the problem of the regulatory gene network reconstruction [3]. Here we demonstrate how the new algorithm provides solutions to problems with several objective functions. The new software developed is available on the project site [4].

2 Methods and Algorithms

2.1 Parallel Differential Evolution

Differential evolution (DE) is an effective stochastic method for function minimization that was proposed by Storn and Price [5]. It starts from a set of the randomly generated parameter vectors q_i, $i = 1, ..., NP$. The set and the vectors are called as population and individuals, respectively. The population on each iteration is referred to as generation. The size of population NP is fixed. The first trial vector is calculated as follows:

$$v = q_{r1} + S(q_{r2} - q_{r3})$$

where $q_{\bullet}$ is the member of the current generation g, S is a predefined scaling constant and $r1$, $r2$, $r3$ are different random indices of the population members.

Being the evolutionary algorithm, DE can be effectively parallelized due to the fact that each member of the population is evaluated individually.

2.2 Enhanced DEEP Method

Like in the original DEEP method [1,2] the "trigonometric mutation rule" [6] is used to take into account a value of the objective function for each individual, and an adaptive scheme for selection of internal parameters is used, based on the control of the population diversity [7] where a new control parameter γ was introduced.

According to the original DEEP method, the whole population is to be divided into branches. The information exchange between branches allowed the best member of the branch to substitute the oldest member of another branch after each Π iterations. It turned out that in biological applications the value of Π is considerably small, making branches an unnecessary complication.

In the Enhanced DEEP method the branches are eliminated, and several oldest members of the population are substituted by the same number of individuals, that have the best values of objective function. In our approach the age of an individual is defined by the number of iterations, in which the individual survived without changes. The number of seeding individuals (i.e. individuals

Algorithm 1. SELECTION

```
proc select (individual) =
{
if ($F <$ the value of the parent) then
   Accept offspring
else
   for all additional criteria $P$ do
      if ($P <$ the value of the parent) then
         Generate the random number $U$.
         if ($U <$ control parameter for this criterion) then
            Accept offspring
         end if
      end if
   end for
end if
}
```

that substitute old ones) Ψ and the number of iterations called seeding interval Θ are the new parameters of the algorithm.

In order to increase the robustness of the procedure we have implemented a new selection rule for DE (Algorithm 1). An offspring is to be accepted to a new generation in accordance with several different objective functions. The offspring replaces its parent if the value of the objective function for the offspring's set of parameters is less than that for the parental one. The additional objective functions are checked in the opposite case. The offspring replaces its parent if the value of any other objective function is better, and a randomly selected value is less than the predefined parameter for this function.

Calculations are finished when the objective function variation is less than a predefined value during the several consecutive steps.

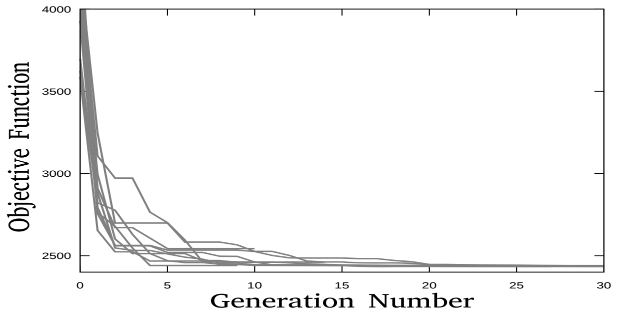


Fig. 1. The combined criterion (1) vs. the generation number for 10 runs. 200 function evaluations were performed by the minimization procedure for each generation.

3 Results

3.1 Implementation

New algorithm was implemented in C programming language as the software package with interface that allows a user to formulate the objective function using different computer languages widely used in biomedical applications, such as Octave, R or KNIME. The control parameters of the algorithm are defined in the datafile that uses an INI-format. The package provides simple command line interface. The DEEP method can be embedded in any new software.

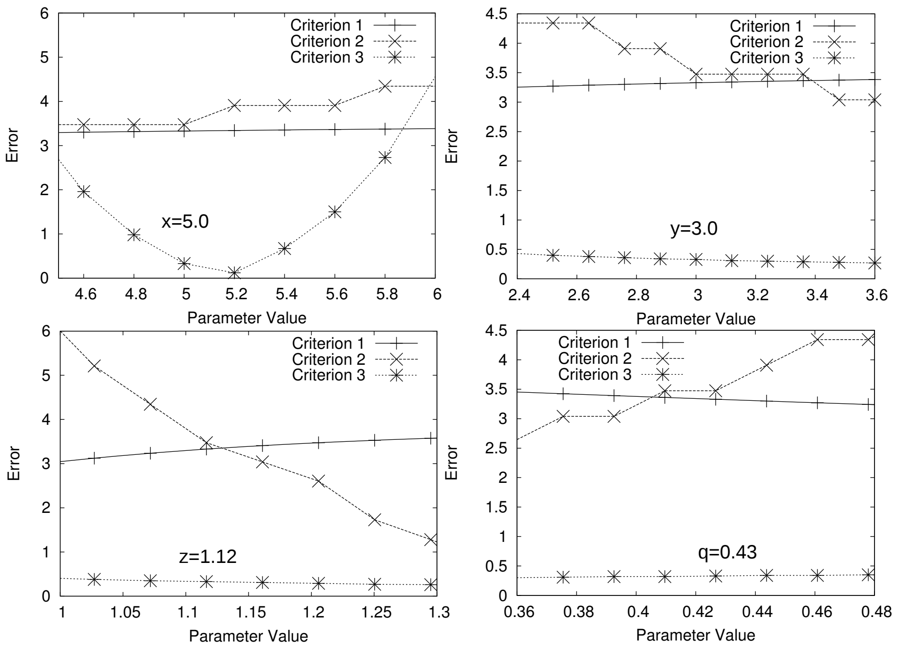


Fig. 2. The criteria graphs in the close vicinity of the optimal values of the four parameters. The values of the parameters found by the algorithm are denoted as x, y, z and q.

One of the parameters of the algorithm determines the number of parallel threads used to calculate the objective function. We utilized the Thread Pool API from GLIB project [8] and constructed the pool with the defined number of worker threads. The calculation of objective function for each trial vector is pushed to the asynchronous queue. The calculation starts as soon as there is an available thread. The thread synchronization condition is determined by the fact that objective function is to be calculated once for each individual in the population and on each iteration.

3.2 The Integer Valued Parameters

The DE operates on floating point parameters, while many of real problems contain integer parameters, e.g., indices of some kind. The following procedure was used to produce the integer vector of parameters from the floating point vector constructed by the DEEP method:

– The values are sorted in ascending order.
– The index of the parameter in the floating point array becomes the value of the parameter in the integer array.

Let us consider the following floating point array of the parameters:

```
a[0] = 0.3; a[1] = 0.5; a[2] = 0.1; a[3] = 0.8;
```

After sorting in the ascending order the array should be transformed to:

```
b[0] = a[2] = 0.1
b[1] = a[0] = 0.3
b[2] = a[1] = 0.5
b[3] = a[3] = 0.8
```

The indices of the sorted array define the current set of the integer valued parameters:

```
i[0] = 2; i[1] = 0; i[2] = 1; i[3] = 3;
```

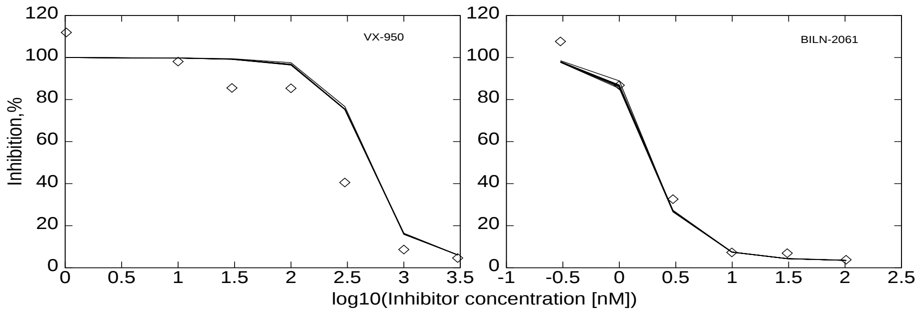


Fig. 3. The viral RNA suppression in the presence of the NS3 protease inhibitors in different concentrations. The dependence of the viral RNA suppression on the increasing concentration of BILN-2061 and VX-950 inhibitors is shown for the third day post-treatment. A solid line is used to show model output and points correspond to the experimental data [11,10].

3.3 The Case Study

We used the new algorithm and software package to determine parameters of the mathematical model of a biological system using experimental data. In biology current experimental methods are unable to measure all the parameters required to describe system behaviour and therefore these parameters are usually to be found by fitting model solutions to data. The model describes the subgenomic Hepatitis C virus (HCV) replicon replication in Huh-7 cells in the presence of the HCV NS3 protease inhibitor [9]. The hepatitis C virus (HCV) causes hazardous liver diseases leading frequently to cirrhosis and hepatocellular carcinoma. No effective anti-HCV therapy is available up to date.

Design of the effective anti-HCV medicine is a very challenging task due to the ability of the hepatitis C virus to rapidly acquire drug resistance. The cells containing HCV subgenomic replicon are widely used for experimental studies of the HCV genome replication mechanisms and the *in vitro* testing of the tentative medicine. These cells are used also to study how the virus acquires resistance to the drugs. HCV NS3/4A protease is essential for viral replication and therefore it has been one of the most attractive targets for development of specific antiviral agents for HCV.

The combined criterion was defined as follows:

$$F_{combined} = F_1 + 0.1 \cdot F_2 + 0.1 \cdot F_3 \tag{1}$$

where the weights were obtained experimentally. The dependence of the best value of the combined criterion in population of individuals on the generation number for 10 runs is plotted in Fig. 1. The objective function is to be evaluated once for each member of the generation, the size of which was set to 200.

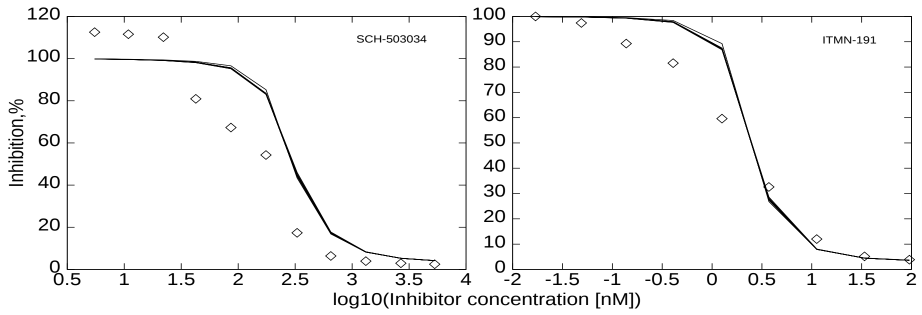


Fig. 4. The predicted kinetics and the suppression rate of the viral RNA in comparison with data not used for parameter estimation. The dependencies of the suppression rate of the viral RNA on the increasing concentration of the SCH-503034 and ITMN-2061 inhibitors. Solid line is used for model output and points correspond to the experimental data [13,12].

The experimental data include kinetic curves of the viral RNA suppression at various inhibitor concentrations of the VX-950 and BILN-2061 inhibitors [11,10].

We seek for the set of parameters that minimizes three criteria. Criterion 1 (F_1) is the sum of squared differences between the model output and data. Additional criteria 2 (F_2) and 3 (F_3) penalize the deviation of the model from a desired behaviour.

The plot of the criteria in the close vicinity of the optimal values of the four parameters from the set is shown in Fig. 2. It is evident that while the criteria do not take a minimum value at one and the same point, nevertheless the algorithm produces reliable approximation of the optimum.

The comparison of model output and experimental dependencies of the viral RNA suppression rate on inhibitor concentration is shown in Fig. 3. As it is clearly seen, the model correctly reproduces experimental kinetics of the viral RNA suppression.

The predictive power of the model was estimated using the experimental data on dependencies of the viral RNA suppression rate on the increasing concentration of the SCH-503034 [13] and ITMN-191 [12] inhibitors. These data were not used for parameter estimation. As can be seen in Fig. 4 the model correctly reproduces experimental observations and thus can be used for *in silico* studies.

Acknowledgments. We thank Ilya Akberdin and Vitaly Gursky for valuable discussions. This work is supported by the State Contract 11.519.11.6041, EU FP7 project 260429 SysPatho and RFBR grant 11-01-00573.

References

1. Kozlov, K., Samsonov, A.: DEEP – Differential Evolution Entirely Parallel Method for Gene Regulatory Networks. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 126–132. Springer, Heidelberg (2009)
2. Kozlov, K., Samsonov, A.: DEEP – Differential Evolution Entirely Parallel Method for Gene Regulatory Networks. Journal of Supercomputing 57, 172–178 (2011)
3. Kozlov, K., Surkova, S., Myasnikova, E., Reinitz, J., Samsonova, M.: Modeling of gap gene expression in Drosophila Kruppel mutants. PLoS Computational Biology 8(8), e1002635 (2012)
4. Differential Evolution Entirely Parallel method, `http://urchin.spbcas.ru/trac/DEEP`
5. Storn, R., Price, K.: Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Technical Report TR-95-012, ICSI (1995)
6. Fan, H.-Y., Lampinen, J.: A Trigonometric Mutation Operation to Differential Evolution. Journal of Global Optimization 27, 105–129 (2003)
7. Zaharie, D.: Parameter Adaptation in Differential Evolution by Controlling the Population Diversity. In: Petcu, D., et al. (eds.) Proc. of 4th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, pp. 385–397 (2002)
8. GLib, `http://developer.gnome.org/glib/`

9. Mishchenko, E.L., Ivanisenko, N.V., Akberdin, I.R., Demenkov, P.S., Likhoshvai, V.A., Kolchanov, N.A., Ivanisenko, V.A.: Suppression of Subgenomic Hepatitis C Virus RNA Replicon Replication In Huh-7 Cells By The NS3 Protease Inhibitor SCH5030334: A Stochastic Mathematical Model. Vestnik VOGIS 16(2), 339–347 (2012) (in Russian)
10. Lin, K., Perni, R.B., Kwong, A.D., Lin, C.: VX-950, a novel hepatitis C virus (HCV) NS3-4A protease inhibitor, exhibits potent antiviral activities in HCv replicon cells. Antimicrob Agents Chemother. 50(5), 1813–1822 (2006)
11. Lin, C., Lin, K., Luong, Y.P., Rao, B.G., Wei, Y.Y., Brennan, D.L., Fulghum, J.R., Hsiao, H.M., Ma, S., Maxwell, J.P., Cottrell, K.M., Perni, R.B., Gates, C.A., Kwong, A.D.: In vitro resistance studies of hepatitis C virus serine protease inhibitors, VX-950 and BILN 2061: structural analysis indicates different resistance mechanisms. J. Biol. Chem. 279(17), 17508–17514 (2004)
12. Seiwert, S.D., Andrews, S.W., Jiang, Y., Serebryany, V., Tan, H., Kossen, K., Rajagopalan, P.T., Misialek, S., Stevens, S.K., Stoycheva, A., Hong, J., Lim, S.R., Qin, X., Rieger, R., Condroski, K.R., Zhang, H., Do, M.G., Lemieux, C., Hingorani, G.P., Hartley, D.P., Josey, J.A., Pan, L., Beigelman, L., Blatt, L.M.: Preclinical characteristics of the hepatitis C virus NS3/4A protease inhibitor ITMN-191 (R7227). Antimicrob. Agents Chemother. 52(12), 4432–4441 (2008)
13. Malcolm, B., et al.: SCH 503034, a mechanism-based inhibitor of hepatitis C virus NS3 protease, suppresses polyprotein maturation and enhances the antiviral activity of alpha interferon in replicon cells. Antimicrob Agents Chemother 50, 1013–1020 (2006)

TCP TIPS: TCP Variant with Proactive Congestion Avoidance

Anatoliy Sivov

Yaroslavl State University, Yaroslavl, Russia
mm05@mail.ru

Abstract. This article introduces TCP TIPS - a TCP variant, which implements a proactive congestion avoidance algorithm. TCP TIPS tries to use the available bandwidth efficiently in both reliable and unreliable networks and simultaneously to decrease the data delivery delay by minimizing queuing delays. Also, this protocol treats other protocols as high priority protocols and yields the required bandwidth share to them, when it is necessary. The article briefly describes algorithms used for TCP TIPS. Some network simulation results with ns-2 are also present in this article.

Keywords: TCP, transport protocols, network congestion, congestion avoidance, network simulation.

1 Introduction

The transport level takes a key place in the hierarchy of network protocols. It provides data exchange between transport service users (i. e. application level protocols) with the required properties (reliable delivery, correct order, etc.). The most widely used transport protocol with reliable data delivery is TCP. This protocol implements the dataflow management algorithm, whose aims are to guarantee a data delivery, a correct packets order and an efficient use of the network capacity. To achieve the last goal TCP tries to set the maximum transmission speed, which does not lead to the network congestion.

The traditional algorithm, used by TCP [1], implements a reactive congestion avoidance method. It uses packet loss detection as an indication of the network congestion. This approach is efficient in reliable networks with routers, whose buffers are correctly sized. However, in the networks with unreliable environment (i.e. wireless networks) or with oversized buffers in the routers TCP can be inefficient or it can result in a significant growth of latency [2].

Proactive algorithms can be used to early detect the network congestion and prevent the latency growth. These algorithms use time characteristics of the dataflow to indicate the congestion. The most widely used characteristic is RTT. TCP Vegas [3] is a classic example of proactive congestion avoidance algorithm implementation.

However, the usage of RTT has several disadvantages. In particular, mechanism of delayed acknowledgments can not be used with these algorithms.

Also, RTT-based algorithms can not distinguish the congestion in the network path used for data delivery from the congestion in the network path used for acknowledgments delivery. Incorrect detection of congestion will result in an underutilization of the network capacity.

Another weakness of many proactive congestion avoidance algorithms and available bandwidth estimation algorithms is related to the use of certain time thresholds (such as BaseRTT). Incorrect choice of the threshold can significantly affect the performance of these algorithms [4].

2 TCP TIPS Algorithms

The motivation for TCP TIPS, the protocol we introduce, is to design a TCP-compatible transport protocol, which efficiently uses the available bandwidth (in reliable and unreliable environments, as well as networks with frequently changing characteristics), has no latency growth issues and issues specific to protocols which use delay-based congestion avoidance algorithms. The other goal is to create a protocol, which treats its traffic as low priority, so that this protocol could be used in common with protocols, that carry realtime data (VoIP, IPTV, and so on). TCP TIPS should yield the required bandwidth share to these protocols' flows and use this share again when it is available.

To achieve these goals TCP TIPS implements a proactive congestion avoidance algorithm, based on the analysis of interpacket time intervals used by a sender and observed by a receiver. This temporal characteristic of dataflow can be used, provided that burst sends are replaced with TCP pacing. With this approach TCP TIPS controls the data transmission speed by changing the values of intersegment intervals on the sender side. The receiver sends an observed value of the time interval between two arrived consecutive segments to the sender with acknowledgment. The sender compares this value with its own interval and decides, how to change the transmission rate. This analysis can be used for congestion avoidance, in particular, due to the fact that an intersegment interval used by the sender is smaller then an intersegment interval observed by the receiver, when the network is congested.

Algorithms, used in TCP TIPS, are described in [6–8]. Some of them are influenced by ARTCP [5]. Briefly, we can divide these algorithms into two items: algorithm for the available bandwidth estimation and dataflow management algorithm.

The first algorithm is based on the fact that a minimum value for the interpacket interval, observed by the receiver in the free network [1], is determined by the network capacity. If the transmission rate is greater than the network capacity, this capacity can be determined in a free network by the following equation:

$$B = \frac{8 \cdot N}{P}, \tag{1}$$

[1] "Free" means here, that there is no other activity in the network.

where B – the network capacity, N – the payload size in bytes, P – the interpacket interval, observed by the receiver.

[6] and [7] show that the following equations are valid, when the total transmission rate of the connections is greater than the network capacity:

$$I = \frac{T \cdot x}{a}, \tag{2}$$

$$P = \frac{T \cdot (a+b)}{a}, \tag{3}$$

where I – the interpacket interval, used by the sender, a – the transmission rate, used by the sender, T – the minimum possible interpacket interval (determined by the network capacity), x – the network capacity, P – the interpacket interval, observed by the receiver, b - the total transmission rate of the other connections.

The sender can change a transmission rate (say, set it to be a' (with corresponding interpacket interval I')), so that the congestion remains. Then it can estimate, what share of the bandwidth it uses and what share is used by the other flows. So, we can get the estimation for the available bandwidth, determined with the following equation:

$$B = a \cdot \frac{I \cdot (a' - a) + a' \cdot (P' - P)}{a' \cdot P' - a \cdot P}, \tag{4}$$

where B – the available bandwidth, P' – the interpacket interval, observed by the receiver, when the sender used the transmission rate a', P – the interpacket interval, observed by the receiver, when the sender used the transmission rate a, I – the interpacket interval, used by the sender to achieve the transmission rate a. [7] shows how to get this equation.

The dataflow management algorithm, used in TCP TIPS, is described in [6, 8]. Its work consists of 4 states: slow start, multiplicative decrease, congestion compensation, and speedup probe/cancel.

After a connection setup TCP TIPS enters a slow start state. It sets an interpacket interval equal to RTT, computed during a connection setup. The sender then changes interval value once per RTT, based on the comparison result of the current interpacket interval used by the sender and the average interpacket interval observed by the receiver since last interval change. TCP TIPS detects the congestion, if the following inequality is true:

$$I < (1 - \varepsilon) \cdot P, \tag{5}$$

where I – the interpacket interval, used by the sender, P – the average interpacket interval, observed by the receiver, when the sender used interval I, ε – a "dead zone" parameter.

If there is no congestion, the sender decreases interpacket interval value, dividing it by $SSGR > 1$ (we use $SSGR = 2$ currently to get the same transmission rate increase as the one that is present in NewReno's "slow start" state). Otherwise, it uses the available bandwidth estimation algorithm to determine the correct transmission rate. TCP TIPS does not use (4) directly. Instead, it uses the same algorithm to determine the quotient ak of the transmission rate to the network capacity and the quotient bk of the total transmission rate of the other connections to the network capacity:

$$ak = \frac{\alpha \cdot P' - P}{I \cdot (\alpha - 1)}, \tag{6}$$

$$bk = \frac{\alpha \cdot (P - P')}{I \cdot (\alpha - 1)}, \tag{7}$$

where $\alpha = \frac{I}{I'}$. [8] explains in detail, how to get these equations.

When ak and bk are determined, the estimated interpacket interval value I_e is set, if it is possible to avoid congestion. This value corresponds to the transmission rate equal to the available bandwidth.

$$I_e = \frac{I \cdot ak}{1 - bk}. \tag{8}$$

After that TCP TIPS enters a multiplicative decrease state. Here, the transmission rate is set lower than I_e (to compensate the congestion) and the compensation area is calculated. Then TCP TIPS enters a congestion compensation state.

The interpacket interval value on congestion compensation is determined by the following equation:

$$I_r = \frac{I_e}{MDF} \tag{9}$$

where MDF - a multiplicative decrease factor, $0 < MDF < 1$.

The compensation area here is determined by the following equation, explained in [8, 6]:

$$S = PRTT \cdot \left(P^{-1} - I_e^{-1} + \sum \left(\frac{SSGR^n}{I'} - I_e^{-1}\right)\right), \tag{10}$$

where $PRTT$ – an average RTT, when the congestion is present. First summand is present only if $I_e > P$. Other summands are present, if $\frac{SSGR^n}{I'} \geq \frac{1}{I_e}$.

TCP TIPS leaves the congestion compensation state after $\Delta t = \frac{S}{I_e^{-1} - I_r^{-1}}$ seconds, where S is determined by (10) or on the moment when new congestion is detected with (5). After that, TCP TIPS enters speedup probe/cancel state. This state is described in detail in [6, 8]. Its role is to check for the available bandwidth increase and to compensate the congestion quickly, if this check fails.

Speedup probe and cancel are determined by (11) and (12), respectively.

$$I_n = min\left\{I', \frac{I}{1 + \beta(I)}\right\}, \tag{11}$$

$$I_n = \frac{I_o}{1 - \beta(I_o)}, \tag{12}$$

where I_n – the new interpacket interval value, I' – the current interpacket interval value, I – the interpacket interval value, that corresponds to the observed interpacket interval P, $\beta(I) = C \cdot I^{3/2}$ – a speedup function, I_o – an interpacket interval value, that corresponds to the transmission rate, when there is no congestion.

3 TCP TIPS Simulation Results

To explore how TCP TIPS behaves in different situations and to analyze its performance and other characteristics, a TCP TIPS simulation model for ns-2 was developed. The main characteristics, that we want to investigate with modeling experiments, are goodput, latency and fairness. The performed experiments can be divided into 5 groups. The first group contains scenarios, that study TCP TIPS behavior in isolated reliable networks. In the second group, a reliable environment is replaced with an unreliable one, so random losses occur. The third group is devoted to study of the TCP TIPS interaction with protocols that use a fixed transmission rate. In the fourth group, data are passed in both directions. And finally, the fifth group contains the experiments, where the network route is changed dynamically, changing RTT or the bandwidth. In all experiments we have a rather simple topology: every route between a sender and a receiver consists of three links and two routers (with DropTail queuing discipline). Host-router links are considered as low-delay highspeed links, but router-router links almost always has a bigger delay and a less capacity.

The modelling in a reliable isolated network shows, that TCP TIPS operates as expected. So, the TCP TIPS transition from a slow start state to multiplicative decrease, congestion compensation and finally to speedup probe/cancel state can be seen. TCP TIPS leaves a slow start soon after the congestion occurred. Multiplicative decrease and congestion compensation results in the release of accumulated on congestion time data from the router's queue. The speedup probe/cancel state keeps the transmission with the correct rate and minimum latency.

In every experiment TCP TIPS got a big goodput (90,5-99,2%) and a low latency (in most cases RTT was very close to the minimum possible RTT for the route). Comparison of the results, obtained with TCP TIPS and several TCP modifications (NewReno, Compound, Highspeed, Illinois, Vegas, Veno, YeAH, CUBIC), shows that TCP TIPS has one of the biggest goodput in most scenarios (up to 10% faster than NewReno in one-way data transmission in a reliable network). In networks with an excessively large router's buffer size TCP TIPS has no latency growth (average RTT fluctuated in the range 254–274 ms, when minimum possible RTT was set to 240 ms) and keeps high goodput (more than 98% in most cases). In networks with an excessive buffering TCP TIPS behaves much better, than NewReno (20-90% bigger goodput), and has the results close to Vegas and YeAH, but has better congestion compensation and lower latency.

In experiments, where several (2–10 connections) TCP TIPS flows coexist, the total goodput fluctuates from 95.36% to 98.46% (up to 40% better than

NewReno, and better than most considered TCP modifications) and fairness for 10 connections is 0.875–0.970 (only YeAH has better fairness results). TCP TIPS also shows good latency results. The biggest average queue utilization was 350 packets (for 10 connections in 90 Mbit/s network), but the queue size was set to 30000 packets. So, TCP TIPS is capable for efficient data transmission with low latency and good fairness results.

The other experiments were made for unreliable networks. Here, we compared goodput results for TCP TIPS, NewReno, Veno, and Westwood+. These experiments displayed that for the loss probability more than 10^{-6} NewReno showed the worst results. Veno concedes to TCP TIPS and Westwood+ for loss probability 10^{-5} and more. Finally, TCP TIPS showed the best results for loss probability more than 0.0001.

The experiments were made, where TCP TIPS coexisted with CBR dataflow. In this case TCP TIPS immediately yields the required bandwidth share, when CBR starts to transmit and takes this share back after CBR dataflow completion. The network congestion, occurred at CBR flow startup, is quickly compensated by TCP TIPS multiplicative decrease.

The other experiment was made to demonstrate that TCP TIPS has no RTT-based protocols disadvantage: inability to distinguish its own congestion and the congestion in the reverse direction (where acknowledgments go). To show this, TCP TIPS and Vegas were tested under the same scenario. Here, they send data in the free network, then on 100th second the CUBIC flow is started in the reverse direction. Due to an excessive buffer size in the routers, this results in a huge RTT growth. Vegas reacts on it as if it is its own congestion and underutilizes the available bandwidth. However, TCP TIPS realizes that there is no congestion in its direction and keeps sending data with a correct rate.

The final group of experiments was devoted to the dynamic change of the route. Here, we made two kinds of experiments: when different routes had different capacity, and when different routes had different delay. These experiments showed that TCP TIPS behaves much better (compared with NewReno or Vegas) on frequent route changes. When the capacity changes, it starts using a new capacity rather fast (correctly compensating the congestion, if there was a capacity decrease). On routes with an equal capacity, but different delays, TCP TIPS, unlike Vegas, has no problem with detecting that the bandwidth has not changed (an average goodput for TCP TIPS in these experiments was more than 97%).

Conclusion. The simulation results show that TCP TIPS exceeds a number of TCP modifications in isolated networks, especially in unreliable environments and on frequent network characteristics changes. TCP TIPS can quickly yield the required bandwidth share to other protocols and take it back on their completion. This was one of protocol design requirements, necessary for using this protocol as low-priority, reliable data delivery service in the networks assigned for high priority realtime data transmission. TCP TIPS does not increase latency and does not obstruct these transmissions. However, TCP TIPS uses the bandwidth efficiently, when it is available.

Acknowledgments. This work was supported by the Russian Foundation for Basic Research grant 12-07-31173 and by The Ministry of education and science of Russian Federation, project 14.132.21.1366.

References

1. Allman, M., Paxson, V., Blanton, E.: TCP Congestion Control. RFC 5681 (2009)
2. Sivov, A.: On Latency Growth in Data Transmission in Communication Networks. All-Russian Competition of Undergraduate and Graduate Students Research in Computer Science and Information Technology: Belgorod, Russia, vol. 2, pp. 292–295 (2012) (in Russian)
3. Brakmo, L., O'Malley, S., Peterson, L.: TCP Vegas: New Techniques for Congestion Detection and Avoidance. In: Proc. ACM SIGCOMM, pp. 24–35 (1994)
4. Hayes, D.A., Armitage, G.: Revisiting TCP Congestion Control using Delay Gradients. In: Domingo-Pascual, J., Manzoni, P., Palazzo, S., Pont, A., Scoglio, C. (eds.) NETWORKING 2011, Part II. LNCS, vol. 6641, pp. 328–341. Springer, Heidelberg (2011)
5. Alekseev, I.V., Sokolov, V.A.: ARTCP: Efficient Algorithm for Transport Protocol for Packet Switched Networks. In: Malyshkin, V.E. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 159–174. Springer, Heidelberg (2001)
6. Sivov, A.: TCP TIPS: Transport Protocol for Unreliable Networks with Latency-Sensitive Data. Modeling and Analysis of Information Systems 19(5), 142–151 (2012) (in Russian)
7. Sivov, A.: Proactive Congestion Avoidance Scheme Based on the Available Bandwidth Estimation. In: Proc. VIII International Research and Training Conference "Current Status of the Natural Sciences and Engineering", Moscow, pp. 108–118 (2012) (in Russian)
8. Sivov, A.: Proactive Network Congestion Avoidance Scheme for Transport Protocol. In: Proc. III International Distance Research and Training Conference "Scholarly Discussion: Problems of Physics, Mathematics and Informatics", Moscow, pp. 116–128 (2012) (in Russian)

The Unified Algorithmic Platform for Solving Complex Problems of Computational Geometry

Vasyl Tereshchenko, Igor Budjak, and Andrey Fisunenko

Taras Shevchenko National University of Kyiv
64/13, Volodymyrska Street, City of Kyiv, Ukraine, 01601
{vtereshch,zerg28,alf.via}@gmail.com
http://www.univ.kiev.ua

Abstract. The problem of complex tasks related to set of points on a plane with help of a generalized and optimized algorithm is considered. The algorithm is easily parallelized. Theoretical analysis has been carried out and practical results have been obtained.

Keywords: unified algorithmic platform, complex tasks, parallel-and-recursive algorithm, computational geometry, weighted concatenable queue.

1 Introduction

At the current moment there exists plethora of algorithms for solving basic tasks of computational geometry. At the same time for solving a complex of tasks there is an opportunity to build a generalized algorithm which uses a common solving concep-tion for all tasks. Visual modeling often requires solving a complex of tasks for the same set of in-put data. Typically the set of input data is processed by each algorithm of the complex sequentially. But such solution can be optimized, if every algorithm will use common data structure and intermediate results of other algorithms. Many of sequential algorithms have reached their theoretical performance limits. And many effective parallel algorithms for solving tasks of computational geometry have already been developed. In particular many of them are described in details in the capacious work [1] containing many references to other authors' results. One can find a description of known methods of constructing convex hull and Voronoi diagram in [2-7]. The task of building a unified parallel algorithm is presented in works [8-10]. The goal of the suggested research is building of a unified algorithmic platform for solving complex tasks of computational geometry.

2 Formulation of the Problem and Solution Method

Problem. Let S be a set of N points in the space R^2. A unified effective parallel algorithm for solving complex of tasks operating under the set S has to be developed with lower complexity estimation $\Omega(NlogN)$ (for single processor computer). An implementation has to provide capability of including other different algorithms for solving tasks of the computational geometry.

2.1 Algorithmic Models

The Algorithmic Model of Existing Systems. If some set of tasks of computational geometry has to be solved then typically a set of separate algorithms is used. The generalized scheme of computations for such approach can be represented as shown on Fig. 1, a. In the worst case these tasks are executed sequentially. For each task one can distinguish the following stages:

1. Preliminary processing – preparing input data and internal data structures which are necessary for work of an algorithm solving the task;
2. Execution of the algorithm – direct solving of target task;
3. Transforming output data for transferring them to the next algorithm if it is necessary.

Such approach obviously creates additional inconveniences for use:transforming data, extra memory usage, complexity in parallelization.

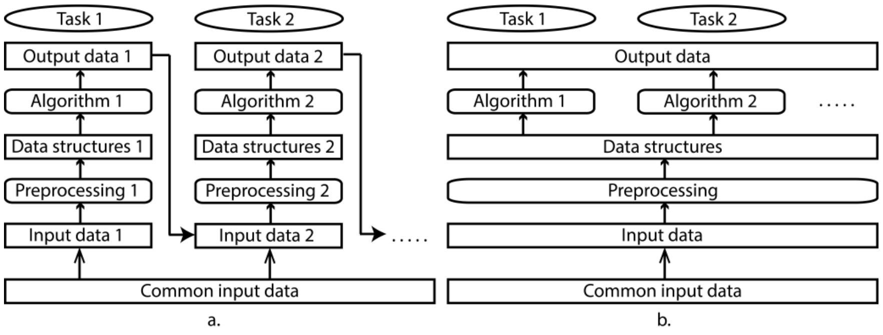


Fig. 1. a) The computational schemes for solving a complex of tasks: a) existing systems; b) the suggested system.

The Algorithmic Model of the Suggested Approach. Unlike the described classical approach we decided to create system having single universal data structure unified for all tasks. Thus, we exempted of the separate data structures and memory consuming. The necessity in preliminary processing procedures for each algorithm in a complex disappears: only one procedure which prepares initial data structure is needed. This leads to execution time reducing. A system should be built such way which enables parallelism on the whole system's level in addition to parallelism on the tasks level. For achieving established goals we decided to use "divide-and-conquer" strategy and concatenable queues as the universal data structure [10]. Thus, we have the following computational scheme presented on Fig. 1, b. Our scheme conventionally is divided into the following stages: *preliminary processing, recursive partitioning, merging subsets' results.*

2.2 The Algorithmic Platform for Solving a Complex of Tasks

Preliminary Processing. As the input we have a set containing N different points. Two sets of points are built: sorted by x and y coordinates. It is known that complexity of sorting on single processor computer is $\Omega(NlogN)$. In our case we can use parallel algorithms of sorting with complexity $O(logN)$ [11].

Recursive Partitioning. The input of this stage is ordered sets of points and as an output we have to obtain AVL-tree which satisfies the following conditions:

1) an order of points during traversal of the tree from left to right must be the same as in the set of input points;
2)each node of the tree must be balanced by the heights of its two sub-trees; difference of their heights must not exceed 1.

Such tree is built by the following algorithm:

1) build tree for one point, constructing one node and storing this point in it;
2) build tree for several points; divide this set of point into two sets which contain the same number of elements (if total number is odd than left side contains one point more). Build node in which the left child is a tree built for the left sub-set and right child is built for the right sub-set. This stage can be parallelized by executing recursive functions on separate processors.

The obtained tree is balanced due to partitioning it into two sub-sets containing equal number of elements.

Merging Results. During merging results for sub-sets two instances of data structures are merged into one. The chosen data structure provides merging in time $\Omega(N)$. Thus total time of algorithm execution is $\Omega(NlogN)$ for a single processor computer.

Interrelation of Algorithms. Let us consider a spectrum of tasks for a set of points on the plane which can be solved with "divide-and-conquer" strategy in time $\Omega(NlogN)$ for a single processor computer. These are such tasks: convex hull; Voronoi diagram; Delaunay triangulation; nearest pair; all nearest neighbors. Interrelation between all intermediate results and input data of algorithms are shown on Fig. 2.

The built implementation allows inserting other algorithms for sets of points which can effectively work using "divide-and-conquer" strategy. Taking into account that all algorithms have common parts of preliminary processing and recursive descend it was decided to subdivide a mechanism of "divide-and-conquer" to a separate procedure containing balanced binary tree and implementing the following scheme:

1. Check if the input set is not trivial;
2. If this is trivial case then execute algorithm for trivial case (e.g. convex hull for 1 point is this point); else go to step 3;
3. Apply algorithm recursively for left and right subsets;
4. Merge results of left and right subsets;
5. Go to the next stage of recursive merge.

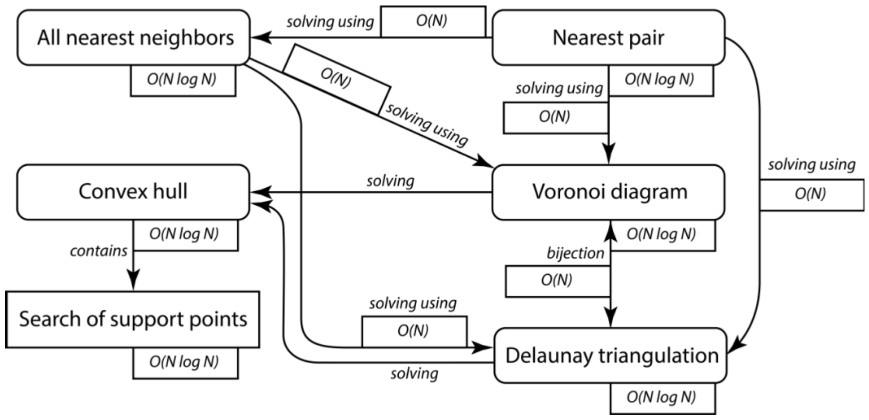


Fig. 2. Interrelation of Basic Algorithms

3 Peculiarities of Parallel Implementation

For parallel implementation the standardized library MPI was used. In particular, it gives flexibility of use and enables cross-platform development. But we note, that MPI is not the only way to implement our algorithm. In the general case time spent on event passing can be estimated as the following:

$$t(S) = t_const + k * |S|, \tag{1}$$

where S – set of transferred data (in our case this is solution for a task for some subset of points) t_{const}- some constant delay caused by network transfer; k – some coefficient which defines time for packing and unpacking.

As it was said above we can parallelize computations on input data partitioning stage with help of recursive descent. For binary tree one can conclude that number of processors should be 2^n, where n – is certain number. While moving down to recursion on its depth n, computations for 2^n processors are run. Each of processors computes tasks on a set with approximate power $\frac{N}{2^n}$ of points, where N – is total number of input points. Sure, we can use any number of processes which is not power of 2. Let total number of processors be $2^n + p$, where $2^n + p < 2^{(n+1)}$. Under such conditions a part processes will solve task for set of points with power $\frac{N}{2^n}$, while another part of processes will solve task for set of power $\frac{N}{2^n+1}$. Such redundancy will not result in performance increase of algorithm.

Thus, execution time for $2^n + p$ processes will be comparable with execution time for 2^n processes. Theoretically it is possible to obtain acceleration on $2^n + p$ processes. For this one should find such number q, which satisfies $p = c * 2^q$. Parallel execution should be run down to depth of recursion $\lceil logN \rceil - q$. With this subtasks are solved for sets of points having power $\frac{N}{2^q}$. The total number of such tasks is $2^{\lceil logN \rceil - q}$. Thus, $2^{\lceil logN \rceil - q}$ sub-tasks are solved on 2^{n+p} processes. The total number of parallel executions is:

$\lceil 2^{\lceil logN \rceil - q}/(2^n + p) \rceil = \lceil 2^{\lceil logN \rceil - q}/(2^n + c * 2^q) \rceil = \lceil 2^{\lceil logN \rceil - q}/(2^q * 2^{n-q} + c * 2^q) \rceil = \lceil 2^{\lceil logN \rceil - q}/(2^q(*2^{n-q} + c)) \rceil = \lceil 2^{\lceil logN \rceil - 2q}/(2^{n-q} + c) \rceil$

As far as $0 < c < 2^{n-1}$, then:

$$2^{\lceil logN \rceil - n - 2q + min(1,q)} \leq \lceil 2^{\lceil logN \rceil - 2q}/(2^{n-q} + c) \rceil < 2^{\lceil logN \rceil - n - q}$$

This way we solve tasks of $\lceil logN \rceil - q$ of recursion depth in time which is less than $2^{\lceil logN \rceil - n - q} * \Omega(\frac{N}{2^q} * log\frac{N}{2^q})$. In the previous case when we use 2^n processes this time is equal to $2^{\lceil logN \rceil - n} * \Omega(\frac{N}{2^q} * log\frac{N}{2^q})$. As soon as $q \geq 0$ we can achieve increase of performance.

4 Conclusions

The conducted research has shown that building a unified algorithmic platform for basic tasks of computational geometry gives possibility to simplify and to speed up solving of complex of tasks for a same set of points. Taking into account that the most of tasks of computational geometry are connected to processing of point sets the suggested approach can serve as a common methodology for wide spectrum of tasks for this discipline. Common merging stages can be unified for other classes of tasks of computational geometry. The only change is specific merge procedures which reflect essence of the concrete task. It was also shown that on practice algorithm is well parallelized and has good performance marks for a parallel environment.

References

1. Aggarwal, A., Chazelle, B., Guibas, L., O'Dunlaing, C., Yap, C.: Parallel computational geometry. Algorithmica 3, 293–327 (1988)
2. Atallah, M.J., Cole, R., Goodrich, M.T.: Cascading divide-and-conquer: A technique for designing parallel algorithms. SIAM J. Comput. 18, 499–532 (1989)
3. Cole, R., Goodrich, M.T.: Optimal parallel algorithms for polygon and point-set problems. Algorithmica 7, 3–23 (1992)
4. Amato, N.M., Goodrich, M.T., Ramos, E.A.: Parallel algorithms for higher-dimensional convex hulls. In: Proc. 35th An. IEEE Sympos. Found. Comput. Sci., pp. 683–694 (1994)
5. Chen, D.: Efficient geometric algorithms on the EREW PRAM. IEEE Trans. Parallel Distrib. Syst. 6, 41–47 (1995)
6. Berkman, O., Schieber, B., Vishkin, U.: A fast parallel algorithm for finding the convex hull of a sorted point set. Internat. J. Comput. Geom. Appl. 6, 231–242 (1996)
7. Goodman, J.E., O'Rourke, J.: Handbook of Discrete and Computational Geometry. Chapman and Hall/CRC Press, N.Y. (2004)
8. Akl, S.G., Lyons, K.A.: Parallel Computational Geometry. Prentice-Hall, Englewood Cliffs (1993)
9. Reif, J.H.: Synthesis of Parallel Algorithms. Morgan Kaufmann, San Mateo (1993)
10. Tereshchenko, V.N., Anisimov, A.V.: Recursion and parallel algorithms in geometric modeling problems. Cybern. and Syst. Analysis 46(2), 173–184 (2010)
11. Cole, R.: Parallel merge sort. In: Proc. 27th IEEE FOCS Symp., pp. 511–516 (1986)

Rigid Body Molecular Dynamics within the Domain Decomposition Framework of DL_POLY_4

Ilian Todorov, Laurence Ellison, and William Smith

Scientific Computing Department, STFC Daresbury Laboratory,
Warrington, Cheshire, WA4 4AD, United Kingdom
ilian.todorov@stfc.ac.uk

Abstract. To remove problematic, high frequency degrees of freedom from a molecular model, modellers often use rigid body dynamics. The method also has additional benefits, for example it allows molecular charge distributions to be conveniently represented by multipolar moments. Rigid bodies are a well established feature within DL_POLY_Classic (formerly known as DL_POLY_2), which employs the replicated data parallelisation methodology. This paper briefly describes the RB formalism and outlines the strategy for its implementation in DL_POLY_4 (formerly known as DL_POLY_3), which uses a very different form of parallelism, namely domain decomposition.

Keywords: Domain decomposition, molecular dynamics, rigid body dynamics, MPI, DL_POLY_4.

1 Introduction

A rigid body (RB) is a collection of point entities whose local geometry is time invariant. One way to enforce this invariance in a simulation is to impose a sufficient number of constraint bonds[1] (CBs) between the atoms in the RB unit. However, for a number of important molecular geometries, use of this method may be either problematic or impossible. Examples in which it is impossible to specify sufficient CBs are linear molecules with more than 2 atoms (e.g. CO_2) and planar molecules with more than three atoms (e.g. benzene). Even when it is possible to define a structure using CBs, the approach suffers from a number of serious, practical disadvantages:

1. The iterative CB solvers/algorithms: SHAKE [1] for leap-frog Verlet (LFV) integration [2] (or RATTLE [3] for velocity Verlet (VV) integration); can be slow to converge, particularly if a ring of constraints is involved, e.g. when one defines water as a constrained triangle.

[1] In DL_POLY_4, CBs are implemented by applying a correction to the forces acting on constraint bonded atoms. The corrected forces, which are determined by means of an iterative procedure, are such that the resulting atom displacements preserve the distances between these atoms within a specified tolerance.

2. It is possible to inadvertently over-constrain a molecule, for example by defining a methane tetrahedron to have 10 rather than 9 bond constraints. In such cases the SHAKE/RATTLE procedure will become unstable.
3. Massless sites (e.g. charge sites) cannot be included in a simple constraint approach, making modelling with potentials such as TIP4P water impossible.
4. The iterative nature of the CB solvers leads to *retrospective*, corrective contributions to the stress and virial, which are out of step with the rest of the system's contributions. This leads to the calculation of inaccurate system pressure.
5. CB solvers lead to positional data updates at each iteration, which requires local interprocessor communications for all neighbouring domains that share constraints. This limits the scalability of the approach when the problem size is fixed.

The alternative to CB dynamics is RB dynamics, which is implemented using the Eulerian equations of rotational motion. The method does not suffer from the drawbacks associated with CBs however, in order to incorporate it into DL_POLY_3 [7,8,6], it was necessary to introduce a new set of data structures as well as a framework for the exchange of this data within the DD scheme.

2 Theoretical Background and Computational Challenges

2.1 Rigid Body Dynamics

The dynamics of RB units which can be described in terms of the translational motion of the centre of mass (COM) and rotation about the COM. To do this we can define the appropriate variables describing the position, orientation and inertia of a RB, and the RB equations of motion.

A RB unit, i, is an assembly of point sites, $j = 1, ..., N_{sites}$, with masses, m_j, and instantaneous velocities, $\underline{v}_j$, and forces $\underline{f}_j$. From this simple description one can define the following quantities:

$$\begin{array}{llll} M_i = \sum_{j=1}^{N_{sites}} m_j & (a) & \underline{R}_i = \frac{1}{M_i} \sum_{j=1}^{N_{sites}} m_j \underline{r}_j & (b) \\ \underline{d}_j = \underline{r}_j - \underline{R}_i & (c) & I_i^{\alpha\beta} = \sum_{j=1}^{N_{sites}} m_j (d_j^2 \delta_{\alpha\beta} - d_j^\alpha r_j^\beta) & (d) \\ \underline{V}_i = \frac{1}{M_i} \sum_{j=1}^{N_{sites}} m_j \underline{v}_j & (e) & \underline{F}_i = \sum_{j=1}^{N_{sites}} \underline{f}_j & (f) \end{array} \quad (1)$$

where M_i is the mass of the RB unit, $\underline{R}_i$ is the location of its centre of mass (COM), $\underline{d}_j$ is the displacement vector of the site j from the COM, $\underline{\underline{\mathbf{I}}}_i$ is the associated rotational inertia matrix, and $\underline{V}_i$ and $\underline{F}$ are its COM velocity and net force. The angular momentum, $\underline{J}_i$, velocity, $\underline{\omega}_i$ and torque, $\underline{\tau}_i$, are easily written as:

$$\begin{aligned} &\underline{J}_i = \sum_{j=1}^{N_{sites}} m_j \left(\underline{d}_j \times \left[\underline{v}_j - \underline{V}_i\right]\right) \quad (a) \qquad \underline{\omega}_i = \underline{\underline{\mathbf{I}}}_i^{-1} \cdot \underline{J}_i \quad (b) \\ &\underline{\tau}_i = \sum_{j=1}^{N_{sites}} \underline{d}_j \times \underline{f}_j \qquad\qquad\qquad (c) \qquad . \end{aligned} \tag{2}$$

The RB's translational motion is treated in a *universal frame* of reference, whereas the rotational motion is calculated in the RB's local reference frame - the so called *principal frame*. This simplifies the equations of rotational motion because, within the principal frame, the rotational inertia tensor, $\hat{\underline{\underline{\mathbf{I}}}}_i$, is diagonal, its components (can be set to) satisfy $I_{xx} \geq I_{yy} \geq I_{zz}$ and are constant.

The orientation of the principal frame with respect to the universal frame can be described via a four dimensional unit vector, the quaternion:

$$\underline{q}_i = [q_0, q_1, q_2, q_3]^T \quad , \tag{3}$$

and the rotational matrix $\underline{\underline{\mathbf{R}}}_i$ to transform from the principal frame to the universal frame is the unitary matrix:

$$\underline{\underline{\mathbf{R}}}_i = \begin{pmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2\,(q_1\, q_2 - q_0\, q_3) & 2\,(q_1\, q_3 + q_0\, q_2) \\ 2\,(q_1\, q_2 + q_0\, q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2\,(q_2\, q_3 - q_0\, q_1) \\ 2\,(q_1\, q_3 - q_0\, q_2) & 2\,(q_2\, q_3 + q_0\, q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{pmatrix} \tag{4}$$

so that if $\hat{\underline{d}}_j$ is the position of an atom in the principal frame (with respect to the COM), its position in the universal frame (also w.r.t. the COM) is given by:

$$\underline{d}_j = \underline{\underline{\mathbf{R}}}_i \cdot \hat{\underline{d}}_j \quad . \tag{5}$$

With all these variables defined, one can now consider the equations of motion for the rigid body unit. The equations of translational motion of a RB are the same as those describing the motion of a single atom, except that the force is the total force acting on the RB, i.e. $\underline{F}_i$, and the mass is the total mass of the rigid body unit, i.e. M_i. These equations can be integrated by the standard Verlet (LFV or VV) algorithms. The rotational motion for a RB is, however, driven by the motion of the torque:

$$\underline{\tau}_i = \frac{d}{dt}\underline{J}_i = \frac{d}{dt}\left(\underline{\underline{\mathbf{I}}}_i \cdot \underline{\omega}_i\right) \quad . \tag{6}$$

This can easily be shown to generate the Euler's equations of rotation for the torque, $\hat{\underline{\tau}}$, and angular velocity, $\hat{\underline{\omega}}$, acting on the body transformed to the principal frame. However, the integration of $\hat{\underline{\omega}}$ is complicated by the fact that as the rigid body rotates, so does the principal frame. Thus a simultaneous integration of the quaternions describing the orientation of the rigid body is required. In DL_POLY_4 this is handled by two different methods - the Fincham Implicit

Quaternion Algorithm (FIQA) [4] for LFV integration and the NOSQUISH algorithm of Miller *et al.* [5] for VV integration. A third scheme, we call *Euler free rotation* [9], is also partially employed when rotation is driven by a force minimisation procedure. Recognising that

$$\frac{d}{dt}\underline{d}_j = \underline{\omega}_i \times \underline{d}_j \tag{7}$$

the torque motion equation (6) can be easily rewritten in terms of the angular velocity update:

$$\frac{d}{dt}\underline{\omega}_i = \underline{\underline{\mathbf{I}}}_i^{-1} \cdot \left(\underline{\tau}_i - \frac{d\underline{\underline{\mathbf{I}}}_i}{dt} \cdot \underline{\omega}_i \right) \quad , \tag{8}$$

which opens the way for solving the equations of rotation explicitly in the laboratory frame. Thus when amended force contributions are generated by a force minimisation procedure, such as a conjugate gradient method, one can easily compute the change in $\underline{\omega}_i$ as driven only by the new torque contributions (which coming from the amended forces acting on the particular RB unit members).

2.2 Implementation within the Domain Decomposition Scheme

The implementation of RB structures and associated dynamics within the DD scheme rests on two working principles. Firstly, in order that inter-domain communications remain local, i.e. only take place between neighbouring cores, the maximum distance between any two sites in a given RB must be less than the largest cut-off distance used in the simulation. This is because the largest cut-off distance determines the width of the halo region surrounding each domain. The restriction thus ensures that if any RB is shared then those parts of it that are not located on the local domain must lie within the domain's halo. It also follows that a given RB may occupy no more than eight domains at any one point in time, see Figure 1.

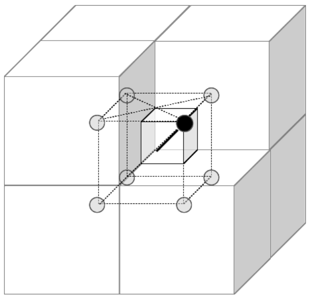


Fig. 1. This sketch illustrates a hypothetical RB consisting of 8 atoms positioned at the corners of a cube. The centre of the cube lies at the voxel of 8 neighbouring domain so that each atom occupies a separate domain.

Secondly, the dynamics of a given RB needs to be calculated on each domain containing at least one site belonging to that RB. All data required for the dynamics therefore need to be distributed to each domain that the RB happens to encroach upon. The dynamics algorithms necessitate careful accounting procedures for shared RB units to prevent multiple counting of RB associated quantities, such as forces, kinetic energy, etc., and to avoid any unnecessary message passing.

In principle, domain haloes include basic particle information only - particles identities and positions, necessary for the evaluation of the forces of all domain particles. However, for particles that are parts of shared RB units it is necessary to extend the level of information haloed to include forces and in some cases velocities. This emerges as a requirement for shared RB units in order to avoid further communications for a number of calculations needed for the dynamics of a RB unit such as COM position, velocity and net force, as well as torque, angular velocity, etc.

In order to calculate the dynamics of a particular RB unit the following information about it is required:

1. The type of unique molecular entity it represents;
2. The list of its constituent sites;
3. The derived quantities: mass per type, COM position and COM velocity and angular velocity;
4. The principal axes and rotational inertia in the principal frame per type;
5. The quaternion vector.

The actual implementation extends the above descriptors with helper arrays such as inverse of mass, rotational inertia and arrays to indicate the RB unit members' status as in-domain (or in-halo), massless, fixed (frozen), etc.

Careful bookkeeping is required for maintaining the data integrity of the model system over all domains at all times. During the dynamics RB units can leave fully a domain or enter a new one. The domain crossing data protocol for a RB is based on that for a particle – the particle is resurrected on the receiving domain and annihilated on the sending domain – with two extra stages. The first stage requires sending to the receiving domain all the unique identifiers of the departing particle (type, index, mass, velocity, etc.) together with its associated topological information (descriptors for chemical bonds, angles, dihedrals, etc.), which will also include the whole set of descriptors associated with the RB unit containing it, before deleting all its data on the sending domain. In the second stage it is determined whether a particle entering a receiving domain belongs to a new or already existing RB unit. In the former case a new set of RB unit descriptors is created and populated with the received data whereas in the latter case an exiting set of descriptors is updated to indicate that the particle is on the domain rather than in the halo. Finally, in the third stage the sending domain removes the descriptors of RB units that have completely left the domain, i.e. with no member particle present. If this step is omitted the number of RB units on a domain may grow to the total number of units in the model system, which means the implementation is no longer memory distributed.

3 Results and Conclusion

Comparative simulations to test the scalability and performance of the RB against the CB implementation in DL_POLY_4 were carried out. Two types of performance tests were used - weak scaling and strong scaling. The weak scaling was carried out on a model system containing 263,424 atoms of equilibrated water (at 300 Kelvin and 1 atmosphere using CBs) in a cubic box with interactions described by the single point charge E (SPC-E) force field. The strong scaling was performed on a larger water system of 444,528 particles, equilibrated at the same conditions, and with the same force field interactions. All runs involved a 100 timesteps of evolution within the velocity Verlet couched microcanonical ensemble (NVE) on processor counts ranging from 16 to 1024 cores on a Cray XE6 platform [10]. All runs used a constant timestep length of 0.5 *femto*-second and a constraint tolerance of 10^{-5} in relative units for the CBs solver [3]. For the weak scaling tests the systems were enlarged by a factor of two every time the core count was increased by a factor 2. Computation time was only measured for the integration algorithms of the two, RB and CB, methodologies and thus excluded start-up and close-down timings. It is worth noting that both integration methodologies, RB and CB, solved the same degrees of freedom per water molecule. In the CB dynamics this is 3×3 (for the three atoms) -3 (for the three distance constraints O-H1, O-H2, H1-H2) or 6 in total. In the RB dynamics this is also 6, 3 for the center of mass translational motion and 3 for the unrestricted rotational motion of the water molecule.

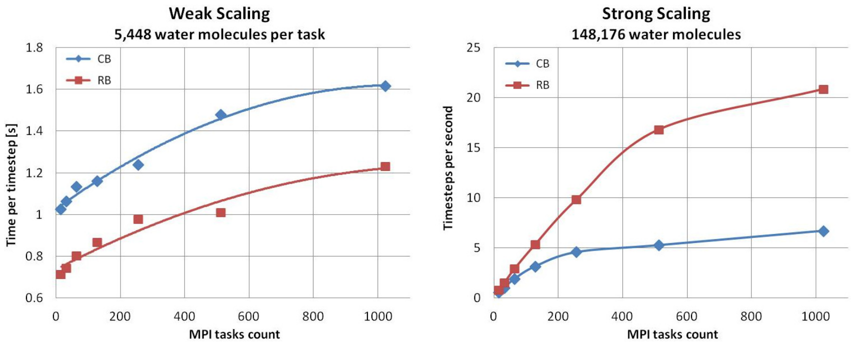


Fig. 2. Performance comparison between CB and RB dynamics as described in the text; (left) weak scaling, (right) strong scaling

Figure 2 presents a performance comparison between RB and CB dynamics methodologies for both scaling tests. It is clear from the weak scaling graph that both RB and CB methodologies scale the same for the tested processor counts range. However, the implicit dynamics solver (CB) is computationally more expensive than the explicit one (RB) by almost a constant amount, ≈ 0.4 sec., of compute time per timestep. The nature of this expense and its consequences are clear from the strong scaling test where, as the core/domain count is increased, DL_POLY_4 performance is pushed from a compute bound mode to a communication bound mode. The CB methodology shows a rapid decay with core count increase as a consequences of the extra local communication and synchronisation required at each iteration of the CB convergence cycle. In contrast, the RB methodology shows excellent scalability over the tested range of cores. Above 512 cores its performance somewhat decays due to a limitations of a part of the DD methodology. Namely, the parallel Verlet neighbour list (VNL) builder which scales linearly with the system size, $O(N)$, where N is the number of particles per core/domain. However, VNL builder performance becomes compromised for $N \leq 500$ when the weight of construction pre-factors starts to dominate over the rest of the algorithm. Further runs, performed to test total system energy conservation (integration stability), showed that the RB integration kept correct dynamics up to timestep lengths of ≈ 1.15 *femto*-second whereas the CB integration could only do that up to ≈ 1 *femto*-second for the selected constraint tolerance.

The inclusion of RB dynamics within the DL_POLY_4 DD framework as an alternative to the CB one using the implementation described in this work introduces a very small computational and communication overheads. These are, however, negligeable with respect to the overheads arising from using CB iterative solvers and do not penalise the scalability of the RB implementation performance as much as they do for the CB even in communication bound regimes. The RB dynamics methodology is computationally faster and superior to CB methodology both in terms of compute time (time per timestep) and discretisation time (size of timestep), as well as being more scalable.

References

1. Ryckaert, J.P., Ciccotti, G., Berendsen, H.J.C.: Numerical integration of the Cartesian equations of motion of a system with constraints: molecular dynamics of n-alkanes. J. Comput. Phys. 23, 327 (1977)
2. Allen, M.P., Tildesley, D.J.: Computer Simulation of Liquids. Clarendon Press, Oxford (1989)
3. Andersen, H.C.: Rattle: a velocity version of the Shake algorithm for molecular dynamics calculations. J. Comput. Phys. 52, 24 (1983)
4. Fincham, D.: Leapfrog rotational algorithms. Molecular Simulation 8, 165 (1992)
5. Miller, T.F., Eleftheriou, M., Pattnaik, P., Ndirango, A., Newns, D., Martyna, G.J.: Symplectic quaternion scheme for biophysical molecular dynamics. J. Chem. Phys. 116, 8649 (2002)
6. The DL_POLY Molecular Simulation Package, `http://www.ccp5.ac.uk/DL_POLY/`
7. Todorov, I.T., Smith, W.: DL_POLY_3: the CCP5 national UK code for molecular-dynamics simulations. Phil. Trans. Roy. Soc. Lond. A 362, 1835–1852 (2004)
8. Todorov, I.T., Smith, W., Trachenko, K., Dove, M.T.: DL_POLY_3: new dimensions in molecular dynamics simulations via massive parallelism. J. Mater. Chem. 16, 1911–1918 (2006)
9. `http://www.ccp5.ac.uk/infoweb/knowledge_center/Euler4.pdf`
10. HECToR: UK National Supercomputing Service, `http://www.hector.ac.uk/`

OPTNOC: An Optimized 3D Network-on-Chip Design for Fast Memory Access

Thomas Canhao Xu, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen

Department of Information Technology, University of Turku, 20014, Turku, Finland
canxu@utu.fi

Abstract. The Network-on-Chip (NoC) paradigm plays an essential role in designing emerging multicore processors. Three Dimensional (3D) NoC design expands the on-chip network vertically. To achieve high performance in a 3D NoC, it is crucial to reduce the access latency of caches and memories. In this paper, we propose an optimized design that provides high performance, low power consumption and manufacturing cost. The proposed scheme shifts the fully connected mesh network to a partially connected network, with the optimization of heterogeneous routers and links. Full system evaluation shows that, compared to a previous optimized heterogeneous design, OPTNOC can further reduce the execution time by 12.1% and energy delay product by 23.5%.

1 Introduction

In recent years, due to the constraint of chip clock frequency and power consumption, microprocessor design is shifting to the concept of Chip Multiprocessor (CMP). To accommodate the increasing number of cores, Network-on-Chip (NoC) was proposed to support the communication among multiple cores [1]. Chip design is moving towards 3D because 2D design has large die size in multiprocessor implementations. 3D integration is a viable solution to increase chip density, providing shorter on-chip wire lengths and faster communication. Through Silicon Via (TSV) [2] has become a promising solution for connecting stacked die layers. They are significantly shorter than wire-bondings, meaning reduced delay and lower power consumption. The manufacturing cost of TSVs is expensive [2], therefore 3D chips are not in volume production yet. Studies have shown the importance of placement of TSVs in a 3D chip [3].

To reduce data access latency, previous works have focused on homogeneous and heterogeneous methods [3] [4]. Mixing buffered and bufferless routers in a NoC is investigated in [5]. In this paper, based on the analysis of application traffic, we propose a novel heterogeneous 3D NoC architecture that improves data access latency by employing synthetic optimization methods. We first investigate how to implement an optimized placement of inter-layer connections and memory controllers. The second optimization focused on heterogeneous non-mesh network, that routers and links are redesigned to improve performance and reduce power consumption. We next explore how performance can be affected by different application mapping and propose an algorithm that maps applications with reduced data access delay and core-core communication delay. To validate our design, we implement our schemes in a full system simulation environment and test the their effectiveness using several applications.

2 Motivation

The performance of a multicore processor is determined by many aspects; for many modern CMPs, improving cache/memory bandwidth and latency is in the highest priority [6]. When a core issues a data request, it checks first if the data is in its private L1 data cache. On an L1 miss, the cache controller will check if another core has the data. If so, the data will be forwarded to the requester. Otherwise, the request is forwarded to the corresponding bank of the shared last level cache. In case the data is modified, according to the cache coherence protocol, snoop messages will be sent to other sharers to invalidate their copies. Finally, if the data is not found in the last level cache, data will be fetched from the main memory. In a cache coherent multiprocessor, the on-chip network transfers different types of messages, e.g. cache coherent messages (control and data) and memory messages. Control messages are usually short packets no more than 64 bits. Assuming 64-byte cache line, the length of data messages can reach 576 bits or 72 bytes. Memory messages sent to the directory controllers are usually the same size of data messages. Considering a 3D NoC with multiple layers, we further classify the messages as inter-layer and intra-layer messages. It is very important to analyse the composition and weight of the on-chip messages. Therefore we examine traces from actual applications running in a 3D NoC (configuration can be found in Section 4).

Results in Figure 1 show that, on average 90.4% of the messages are core-cache communication, meaning these messages are transmitted between layers. Core-core communication occupy 3.9% of on-chip traffic, while the rest 5.8% traffic are cache-cache communication. Some applications, e.g. FFT and Ocean, exhibited higher core-core communication (around 6%). While Raytrace and Water-Nsq show less than 2% core-core communication. The results implied that core-cache communication dominated the on-chip traffic, while core-core communication is negligible. Based on this insight, we propose an optimized 3D NoC design.

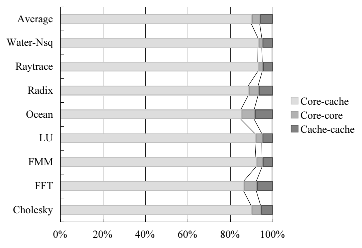


Fig. 1. Composition of on-chip messages

3 The OPTNOC Design

OPTNOC is a low-cost 3D NoC design optimized for fast memory access. Based on the aforementioned analysis of traffic trace, OPTNOC leverage several insights to provide high performance memory access.

3.1 Optimal Placement of Cache and Memory Controller

Here we study an 8×8 3D NoC with two layers (details can be found in Section 4). Notice that, more cache layers can be stacked in the 3D NoC. Figure 2 illustrated the implementation of pillars [3] [7]. As illustrated, only a quarter of the nodes are connected by vertical pillars (A pillar *P* is defined as a bunch of TSVs, including TSVs for data, control and power distribution). When a core issues a request, the nearest pillar is selected to access the other layers. If there are two adjacent pillars, the pillar in the X direction will be chosen. Arrows in Figure 2 show the routing for data flow. X-Y deterministic routing algorithm is used as the message reaches the cache layer. Obviously, in Figure 2, all cores can reach the pillar within one hop. The problem of optimal placement of memory controllers has been studied in [7]. It is shown that an system performance will be affected by different placements of memory controllers. Here we apply the optimal placement, which is the same as pillars.

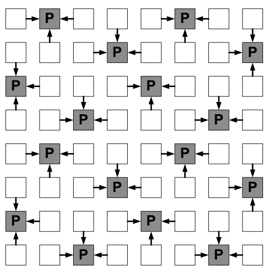


Fig. 2. Optimal placement of pillars and routing to pillar

3.2 Heterogeneous Architecture

Heterogeneous NoC architectures have been proposed in many papers, with different implementation approaches [4] [8]. Here we apply a unique method based on the aforementioned observations, i.e. direct core to core communication in a 3D NoC is very rare. Routers of most nodes are removed, those nodes are connected to the router of the pillar node directly. Figure 3 illustrated nodes in the upper right corner. Here, only one node in four has a router, while other three nodes connect to the router via direct links (solid line routers and links). In this case, all *Pillar Routers* connect four cores and the other layer(s). Comparing with traditional homogeneous design, e.g. right side of Figure 3, the proposed approach saves area, decreases delay of intermediate routes and reduces power consumption. For example, since node 54 has no router, router delay is eliminated accordingly. Accessing the cache layer is faster in OPTNOC as well due to the removed routes. However, a major problem of our proposed architecture is that, there is no direct connection between several cores. For example, cores 54 and 62 are not connected, a message from core 54 has to go through node 55 to the next layer,

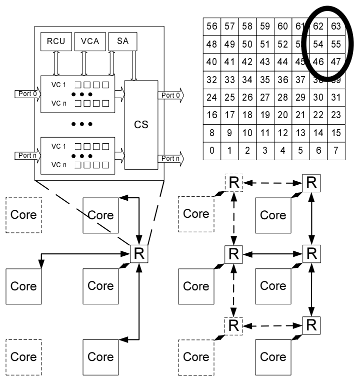


Fig. 3. Router and link of nodes. The Router R consists of a Routing Computation Unit (RCU), a Virtual Channel Allocator (VCA), a Switch Allocator (SA), a Crossbar Switch (CS), several Virtual Channels (VC) and input buffers.

then route to node 61, afterwards the destination node. However, the direct communication between cores are very rare.

Another optimization is introduced to mitigate the drawback of overall performance. The flit width of modern NoCs is typically 128 bits, hence control messages can be encoded to one flit, while data messages can be encoded to five flits. Flit width determines link width, crossbar size and buffer size, therefore the power consumption of routers and links, as well as the network performance are affected. Heterogeneous designs of routers/links have been explored in several papers [4] [8]. In OPTNOC, we implement only one type of 16 big router in the pillar nodes. The links connecting big routers are widened as well (288 bits of flit width). In this situation the data messages can be transferred in two flits, and four control messages from cores can be combined. Assuming 3mm wire length, total link area and power consumption of OPTNOC are 3.6% lower than the homogeneous 128 bits design. We model routers under 32nm processing technology, with 3.0GHz operating frequency and 1V voltage. Simulation result shows that, even with additional logic, the total router area of OPTNOC is 16.48mm^2, 45.2% less than homogeneous. Total power consumption of routers in OPTNOC is lower as well. We note that the power consumption for transferring a control message from core to cache is higher in OPTNOC comparing with conventional architecture. However multiple concurrent messages can be combined to improve channel utilization and reduce overall power consumption. In addition, transferring data messages consumes less power due to reduced fragmentation.

3.3 Application Mapping

In the OPTNOC design, since cores have heterogeneous connections, mapping application to specific regions in the network can achieve better performance. Here we propose a mapping algorithm that maps application processes/threads close to the same pillar router if possible. The scheduling algorithm takes several metrics into account: first, the Average Cache access Latency (ACL) is calculated; second, it tries to allocate cores that connected with same pillar router; third, the communication between cores should be minimized. For ACL, since the cache slices are shared by all cores, and data are mapped to cache slices according to their addresses, we calculate the average hop counts for a core accessing the shared cache slices. Obviously, nodes in the corners of the NoC have much higher ACL than nodes in the center. However, nodes directly connected with a pillar usually have lower ACL, sometimes even lower than inner nodes. As the task request more cores, other metrics are considered. In OPTNOC, a pillar router connects four cores, therefore a task with four or less processes/threads will be mapped to the cores connected by the same router. If a task requests more than four cores, adjacent cores with lower ACL will be considered first.

4 Experimental Evaluation

4.1 Experiment Setup

We evaluate and compare OPTNOC with homogeneous 3D NoC design and other heterogeneous designs. Our simulation platform is based on a cycle-accurate

NoC simulator (GEMS/Simics [9] [10]). We implement OPTNOC with two layers. The processor layer is an 8×8 mesh of 64 Sun UltraSPARCIII+ cores with private L1 cache (split I + D, 16KB + 16KB, 4-way associative, 64-byte line, 3-cycle access delay). The cache layer consists of 16MB shared L2 caches, divided into 64 banks/slices, each 512KB. Simulations are run on Solaris 9. Orion2 power simulator for interconnection on-chip networks is used to evaluate detailed power characteristics of routers and links. Workloads used here are selected from SPLASH-2, SPECjbb and TPC-H [11] [12] [13].

4.2 Performance Analysis

Here we provide a performance analysis of OPTNOC architecture with different applications. In particular, we compare the performance of OPTNOC with a homogeneous 3D NoC design [3] (*Homo* in figures), a heterogeneous design [4] (*Hetero-M* in figures), and an optimized heterogeneous design [8] (*Hetero-X* in figures, we compare the 96/192 design). To further verify the effectiveness of our application mapping algorithm, we classify the OPTNOC into two groups: one with proposed mapping algorithm (*OPTNOC* in figures), the other with arbitrary mapping (*OPTNOC-A* in figures, here the mapping algorithm is from Solaris 9). All these designs are modified with same running environment, e.g. frequency and flit width, to provide comparable results. We measure performance in terms of execution time and energy delay product.

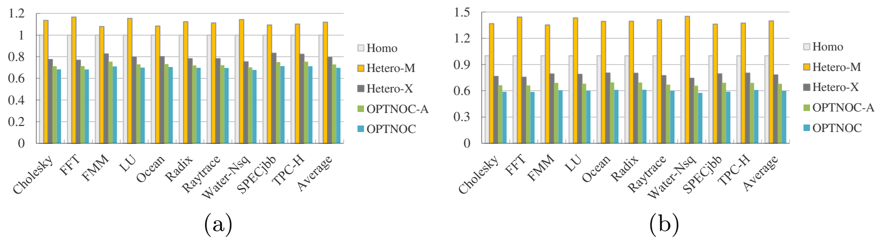


Fig. 4. Normalized execution time (a) and energy delay product (b) with OPTNOC and other NoC designs

The results in Figure 4a show that, the OPTNOC outperforms other architectures in terms of execution time. For example, even without optimized application mapping, the average execution time of applications of OPTNOC is still 27.1% and 8.2% lower than *Homo* and *Hetero-X* respectively. The proposed application mapping algorithm further provides 4.2% performance improvement. This is primarily due to the partially connected network, since the processing capability of routers and links in the hot-spot areas are increased, and components with low utilization are removed. Throughput of data messages is improved significantly over wide routers and links as these messages can be transmitted within two flits in OPTNOC. We note that applications with higher network injection rate take more advantage from OPTNOC, while applications with higher core-core communication benefit more from optimized application mapping.

In terms of energy delay product, the OPTNOC shows improved values than other designs (Figure 4b). On average, the energy delay product of OPTNOC without mapping optimizations (*OPTNOC-A*) is 32.1% and 13% lower than *Homo* and *Hetero-X*, respectively. As mentioned before, we calculate the overall access latency of data requests and responses. Here, reduced hop counts to cache nodes and memory controllers in OPTNOC contribute to the performance improvement. On the other hand, reduced number of routers and links in OPTNOC decreases overall power consumption from the on-chip network. We note that proper application mapping provides 12% better energy delay product. The reduced execution time translated into lower network latency and power consumption. It is also noteworthy that OPTNOC provides both higher performance and higher efficiency than the two designs proposed in [8], meaning that our optimizations are effective.

5 Conclusion

We proposed OPTNOC, an optimized 3D NoC design in this paper. We analyzed cache/memory access latencies in an on-chip network. Based on traffic profile from eight applications, we discovered that direct core-core communication was relatively low in comparing with other traffic. Hence in this paper, a novel 3D NoC design was introduced. The mesh network was redesigned to fit the characteristic of application traffic. Full system experiments have shown that, comparing with an earlier 3D NoC design, the average execution times and energy delay product were reduced by 12.1% and 23.5% respectively.

References

1. Dally, W.J., et al.: Route packets, not wires: on-chip inteconnection networks. In: Proceedings of the 38th DAC, pp. 684–689 (June 2001)
2. Velenis, D., et al.: Impact of 3d design choices on manufacturing cost. In: IEEE 3DIC 2009, pp. 1–5 (September 2009)
3. Xu, T., et al.: Optimal number and placement of through silicon vias in 3d network-on-chip. In: Proceedings of the 14th DDECS, pp. 105–110 (April 2011)
4. Mishra, A.K., et al.: A case for heterogeneous on-chip interconnects for cmps. In: Proceedings of the 38th ISCA, pp. 389–400 (2011)
5. Zhao, H., et al.: Exploring heterogeneous noc design space. In: Proceedings of the 2011 ICCAD, pp. 787–793 (2011)
6. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation (2013)
7. Xu, T.C., et al.: Optimal memory controller placement for chip multiprocessor. In: Proceedings of the 8th CODES+ISSS, pp. 217–226 (October 2011)
8. Xu, T.C., et al.: A high-efficiency low-cost heterogeneous 3d network-on-chip design. In: Proceedings of the 5th NoCArc, pp. 37–42 (December 2012)
9. Magnusson, P., et al.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002)
10. Martin, M.M., et al.: Multifacet's general execution-driven multiprocessor simulator (gems) toolset. Computer Architecture News (September 2005)
11. Woo, S.C., et al.: The splash-2 programs: Characterization and methodological considerations. In: Proceedings of the 22nd ISCA, pp. 24–36 (June 1995)
12. SPEC: Specjbb 2000, `http://www.spec.org/jbb2000/`
13. TPC: Tpc-h decision support benchmark, `http://www.tpc.org/tpch/`

Author Index