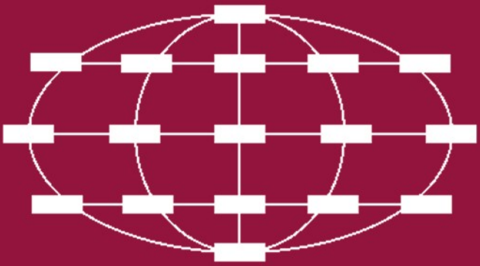
Victor Malyshkin (Ed.)

LNCS 5698

Parallel Computing Technologies

10th International Conference, PaCT 2009
Novosibirsk, Russia, August/September 2009
Proceedings



Springer

Victor Malyshkin (Ed.)

Parallel Computing Technologies

10th International Conference, PaCT 2009
Novosibirsk, Russia, August 31-September 4, 2009
Proceedings

Springer

Volume Editor

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

Library of Congress Control Number: 2009931640

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

ISSN	0302-9743
ISBN-10	3-642-03274-5 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-03274-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12724918 06/3180 5 4 3 2 1 0

Preface

The PaCT-2009 (Parallel Computing Technologies) conference was a four-day event held in Novosibirsk. This was the tenth international conference to be held in the PaCT series. The conferences are held in Russia every odd year. The first conference, PaCT 1991, was held in Novosibirsk (Academgorodok), September 7–11, 1991. The next PaCT conferences were held in Obninsk (near Moscow), August 30 to 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. Since 1995 all the PaCT Proceedings have been published by Springer in the LNCS series. PaCT-2009 was jointly organized by the Institute of Computational Mathematics and Mathematical Geophysics of the Russian Academy of Sciences (RAS) and the State University of Novosibirsk. The purpose of the conference was to bring together scientists working on theory, architecture, software, hardware and the solution of large-scale problems in order to provide integrated discussions on parallel computing technologies. The conference attracted about 100 participants from around the world. Authors from 17 countries submitted 72 papers. Of those submitted, 34 were selected for the conference as regular papers; there were also 2 invited papers. In addition there were a number of posters presented. All the papers were internationally reviewed by at least three referees. A demo session was organized for the participants. Different tools were submitted for demonstration and for the tutorial, one of them being WinAlt (Windows Animated Language Tool) for description. We would like to extend many thanks to our sponsors: the Russian Academy of Sciences, the Russian Fund for Basic Research, IBM, NVIDIA, HP and Microsoft for their financial support.

September 2009 Victor Malyshkin

Organization

PaCT 2009 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 Novosibirsk National University and Novosibirsk State Technical Universities.

Organizing Committee

Conference Chair	Victor Malyshkin (Russian Academy of Sciences)
Conference Co-chair	Mikhail Lavrrenliev (Novosibirsk National University, Russia)
Conference Secretary	Maxim Gorodnichev (Russian Academy of Sciences)
Organizing Committee	V. Malyshkin (Russian Academy of Sciences)
	M. Lavrentiev (Novosibirsk National University, Russia)
	B. Glinskiy (Russian Academy of Sciences)
	M. Gorodnichev (Russian Academy of Sciences)
	S. Achasova (Russian Academy of Sciences)
	K. Chajuk (Russian Academy of Sciences)
	S. Kireev (Russian Academy of Sciences)
	V. Perepelkin (Russian Academy of Sciences)
	A. Usov (Russian Academy of Sciences)

Program Committee

V. Malyshkin	Russian Academy of Sciences (Chairman)
S. Abramov	Russian Academy of Sciences
S. Bandini	University of Milano-Bicocca, Italy
O. Bandman	Russian Academy of Sciences
F. Cappello	INRIA, France
T. Casavant	University of Iowa, USA
B. Chetverushkin	Russian Academy of Sciences
P. Degano	State University of Pisa, Italy
D. Désérable	INSA, Rennes, France
S. Gorlatch	University of Münster, Germany
Yu. Karpov	St. Petersburg Polytechnical University, Russia
K.-C. Li	Providence University, Taiwan
T. Ludwig	Ruprecht-Karls-Universität Heidelberg, Germany
G. Mauri	University of Milan, Italy
M. Valero	Barcelona Supercomputer Center, Spain
D. Petcu	Western University of Timisoara, Romania

M. Raynal	IRISA, Rennes, France
B. Roux	IRPHE, France
P. Sloot	University of Amsterdam, The Netherlands
C. Trinitis	LRR, Munich, Germany
R. Wyrzykowski	Czestochowa University of Technology, Poland
L. Yang	St. Francis Xavier University, Canada

Referees

S. Gorlatch
F. Glinka
P. Kegel
A. Ploss
M. Schellmann
O. Bandman
V. Malyshkin
M. Gorodnichev
S. Kireev
Y. Karpov
T. Ludwig
P. Degano
M. Raynal
A. Khutoretskij
D. Petcu
T. Casavant
A. Nepomniaschaya
S. Achasova
D. Désérable
G. Mauri

Sponsoring Institutions

Russian Academy of Sciences
The Russian Fund for Basic Research
IBM
NVIDIA
HP
Microsoft

Table of Contents

Asynchronous Language and System of Numerical Algorithms Fragmented Programming

Sergey Arykov and Victor Malyshkin

Supercomputer Software Department,
Institute of Computational Mathematics and Mathematical Geophysics
Russian Academy of Sciences,
6, pr. Lavrentieva, 630090, Novosibirsk, Russia
{arykov,malysh}@ssd.sscc.ru

Abstract. A fragmented approach to parallel programming of numerical methods and its implementation in the asynchronous programming system Aspect are considered. It provides several important advantages like automatic implementation of dynamic properties (setting up on available resources, dynamic load balancing, dynamic resource distribution, etc.) of an application program. The asynchronous parallel programming system Aspect is considered which implements a conception of fragmented programming on supercomputers with shared memory architecture.

Keywords: fragmented technology of programming, asynchronous languages and programming systems, dynamic program's properties, automation of parallel realization of numerical models.

1 Introduction

The development of high-quality parallel programs still remains a complicated task because the developer has to manually solve many various problems related to dynamic properties of a parallel program (dynamic setting up on available resources, dynamic load balancing, dynamic resource distribution, etc.).

In this paper we propose a fragmented approach to parallel numerical programs development and organization of computations [1-2], which allows to provide an automatic implementation of dynamic properties of an application program. The main idea of this approach is to represent a program as a set of computational fragments, which are sufficiently small for loading available resources but still big enough to keep the overhead from control reasonable.

2 A Fragmented Approach to Parallel Programming

The essence of the fragmented approach is to represent an algorithm and its implementing program as a set of *data fragments* and *code fragments*. In the course of execution, the fragmented structure of a program is kept.

Each code fragment is supplied with a set of *input data fragments* used to compute *output data fragments*. The substitution of data fragments as parameters into a code fragment is referred to as *applying* a code fragment to data fragments (the same code fragment may be applied to different data fragments). The code fragment with its input and output data fragments constitutes a *computation fragment*. On the set of computation fragments a partial order (*control*) is defined. The resulting program is created from such computation fragments, with fragmentation of the program kept during the program execution.

Execution of a fragmented program is the execution of computation fragments in any order that does not contradict to the defined control. Each computation fragment receives its resources during setting on execution, creates a new process of the program and can migrate from one processor to another.

Consider a simple example of matrix multiplication algorithm. Three 9 x 9 matrices *A*, *B* and *C* are given. Matrix *C* is initialized with zeros.

First, we need to construct the matrices from a number of 3x3 submatrices (Fig. 1). Each submatrix represents a data fragment which is denoted with the name of the matrix along with the number of the submatrix. For example, $C_{(1,1)}$ denotes the first fragment of the matrix *C*.

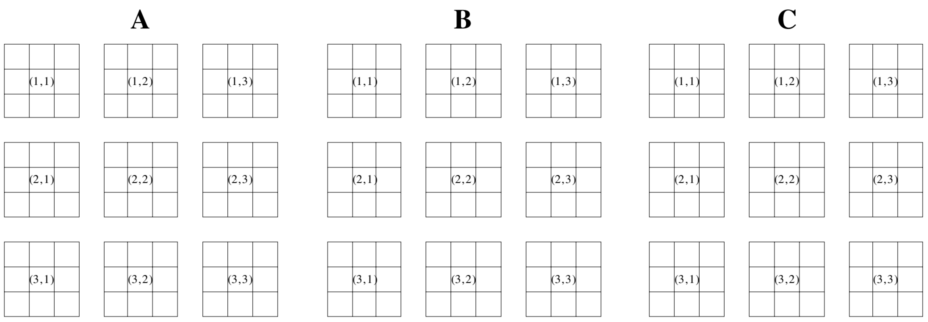


Fig. 1. Fragmentation of the matrix multiplication task

Then, we define a code fragment *F*, which receives two data fragments as an input and produces one data fragment as an output by multiplying the first data fragment by the second one according to formula (1):

$$c_{ij} = \sum_k a_{ik} b_{kj} \tag{1}$$

(here c_{ij} is an entry in the resulting data fragment). Then the product of the matrix *A* and the matrix *B* multiplication can be calculated using formula (2)

$$C_{(i,j)} = \sum_k A_{(i,k)} B_{(k,j)} \tag{2}$$

(here $C_{(i,j)}$ is a data fragment of the matrix *C*). Addition of the results can either be implemented as a separate code fragment or embedded into the code fragment *F* (as is offered in the sequel).

Now let us apply the code fragment to appropriate data fragments, i.e. define the computation fragments. We denote the computation fragment with the name of the code fragment followed by a list of data fragments in parentheses, which are used by this code fragment. To compute $C_{(1,1)}$ it is required that three computation fragments be executed: $F(A_{(1,1)}, B_{(1,1)}, C_{(1,1)})$, $F(A_{(1,2)}, B_{(2,1)}, C_{(1,1)})$ and $F(A_{(1,3)}, B_{(3,1)}, C_{(1,1)})$. Since each of those fragments writes the results obtained to the data fragment $C_{(1,1)}$, we should define the order of executing the computation fragments in order to avoid the data racing. For example, we can require that $F(A_{(1,2)}, B_{(2,1)}, C_{(1,1)})$ be executed after $F(A_{(1,1)}, B_{(1,1)}, C_{(1,1)})$, and $F(A_{(1,3)}, B_{(3,1)}, C_{(1,1)})$ after $F(A_{(1,2)}, B_{(2,1)}, C_{(1,1)})$. Other computation fragments to compute the rest of the matrix *C* can be formed in a similar way, and as a result we obtain a fragmented program. It should be noted that the computation fragments that compute different data fragments of the matrix *C* can be executed in parallel.

The example above brings about a few important conclusions:

1. The algorithm used inside the code fragment differs from formula (1) in an extra operation of addition. A difference is not random as the fragmentation of the algorithm can require its modification and so it cannot be accomplished automatically.
2. The algorithm can be fragmented in different ways. For example, for an 8x8 matrix data fragments 2x2, 2x4, 4x4, etc. can be used. The only important thing is that it should be possible to construct the initial data structures from all data fragments. The choice of the size of data fragments directly affects the overall number of the fragments in the program being the major parameter of the program fragmented.

Algorithms of different application areas can be fragmented with a different quality. In this paper, we consider the fragmented approach as applied to realization of numerical models. In this area, the approach provides considerable benefits:

1. Ability to automatically construct a parallel program. When the fragmented approach is applied, the most complex dependences are hidden inside code fragments, which allow a certain formalization of constructing a parallel program based on the control scheme.
2. Ability to provide a parallel program with a set of dynamic properties. It can be attained due to the fragmented structure of a program.
3. Portability between different architectures. As a fragmented program allows some flexibility in choosing the order of computation fragments execution, there is a good reserve for adopting the program to a specific supercomputer.
4. Possibility to store control schemes for different tasks in the library for their further reuse.

3 Implementation of the Parallel Programming System Aspect

The parallel programming system Aspect is one of possible implementations of the fragmented approach to developing parallel programs. At the moment it is oriented to finding solutions for numerical models using multiprocessors and multicores architectures.

The Aspect is based on asynchronous model of computation. The system consists of the two main components: a translator and an executive subsystem. The input for the translator is a text written in the Aspect programming language; the output is an asynchronous program in C++. After translation, the asynchronous program is combined with an executive subsystem, written in C++, and is compiled into an execution file. It is done using a standard C++ compiler, e.g., gcc.

The Aspect programming language allows representing algorithms with different levels of asynchronism. Its main peculiarities are:

1. Static data types.
2. Explicit declaration of data dependences between operations.
3. Partial distribution of resources (it allows several assignments to one variable).
4. Focus on regular data structures (the main data type is an array).
5. An imperative language (like C++) is used to define computations inside code fragment.

A detailed description of Aspect programming language is out of the scope of this paper, thus its main properties will be shown in a simple task of matrix multiplication discussed in Section 2.1.

The text of the program solving the task of matrix multiplication on the Aspect programming language.

```
program MultMatrix {
data fragments
  double Matrix[m][m]
code fragments
  F(in Matrix A, Matrix B, Matrix C; out Matrix C) {
    for(int i=0; i<m; i++)
      for(int j=0; j<m; j++)
        for(int k=0; k<m; k++)
          C[i][j] += A[i][k]*B[k][j];
  }
task data
  Matrix A[n][n], B[n][n], C[n][n]
task computations
  S[i][j][k]: F(in A[i][k], B[k][j], C[i][j]; out
      C[i][j]) where i: 0..n-1, j: 0..n-1, k: 0..n-1
task control
  S[i][j][k] < S[i][j][k+1]
}
```

The meaning of most constructions of the program is obvious. Computations inside the code fragment *F* are defined using C++ language. In the section *task computations* application of the code fragment *F* to the task data is defined. Each of the indices *i*, *j*, *k* goes through all values in the range defined after the index name; each combination of (*i*, *j*, *k*) creates a separate computation fragment. In the section *task control* we define the order of executing different computation fragments. The computation fragments with equal indices *i* and *j* will be executed sequentially as index *k* grows.

4 Results of Experiments

As a test platform, the computer with the following configuration was used: Athlon 64 X2 3600+ (2*256 L2) / 1024 DDR2 / Windows Vista Home Premium (32 bit) / Visual C++ 9.0 (with options /O2 and /arch:SSE2).

All tasks were implemented in C++. Entries of the matrices are real numbers with double precision. In all the tables, the time of computing is given in seconds.

4.1 Matrix Multiplication

Fragmentation of this algorithm is described in Section 2.1.

The 'Standard solution' method is implemented according to formula (1). When solving this task with ACML, the 'dgemm' procedure was used. Two versions of the fragmented approach were tested: one using formula (1) for computations inside the code fragment ('standard solution') and the other one using the 'dgemm' from ACML library for computations inside the code fragment ('ACML'). The results obtained are shown in Table 1.

Table 1. Matrix multiplication task

Method/Matrix size	512x512	1024x1024	2048x2048
Standard solution	3,26	45,64	499,22
AMD Core Math Library 4.2.0	0,09	0,73	5,67
Fragmented approach (standard solution)	0,22	1,78	14,52
Fragmented approach (ACML)	0,16	1,00	7,89

A fragmented version of the matrix multiplication algorithm is about 25 times as fast as the standard solution, since the fragmented approach using cache memory is much more efficient. At the same time, it is about 2.5 times worse than ACML because ACML is optimized especially for AMD on a low level). However, if we use the 'dgemm' inside the code fragment, then the difference will come as little as 40%. That is a good result with allowance for all advantages that the fragmented approach automatically provides to a parallel program.

4.2 LU Decomposition

An algorithm can be fragmented in the following way [3]: the source matrix is built out of the data fragments similar to the matrix multiplication task (see Fig. 1) and four code fragments are created. The first code fragment will process the data located on the main diagonal; the second code fragment – the data fragments to the right of the first data fragment; the third code fragment – the data fragments below the first data fragment and the fourth code fragment will calculate the rest data fragments.

Computations are performed through iterations. On the first iteration the data fragment $A_{(1,1)}$ will be computed. After that, the data fragments to the right ($A_{(1,2)}$, $A_{(1,3)}$) and below it ($A_{(2,1)}$, $A_{(3,1)}$) can be computed simultaneously. Finally, the internal matrix ($A_{(2,2)}$, $A_{(2,3)}$, $A_{(3,2)}$, $A_{(3,3)}$) should be recomputed. The next iteration will be applied only to the internal matrix, thus $A_{(2,2)}$ will be first computed, then $A_{(2,3)}$, $A_{(3,2)}$ and, finally, $A_{(3,3)}$, etc.

Two versions of the standard solution were tested with a column and a row order of the arrays representation in the memory. When solving this task with ACML, the 'dgetrf' procedure was used. Our results are shown in Table 2.

Table 2. LU decomposition task

Method/Matrix size	**512x512**	**1024x1024**	**2048x2048**
Standard solution (column order)	2,55	24,98	244,06
Standard solution (row order)	0,27	2,14	16,30
AMD Core Math Library 4.2.0	0,03	0,26	2,11
Fragmented approach (column order)	0,08	0,67	5,52
Fragmented approach (row order)	0,08	0,66	5,23

We see that the difference in time for the standard solution with different order of entries in the memory is multiple to ten, but with the fragmented approach it becomes inessential. That is another benefit of the fragmented approach: even when the user does not know anything about peculiarities of the arrays representation in the memory, the fragmented structure of the program will not allow losing much in performance.

In the task in question, the algorithm fragmentation requires some changes in it, which do not allow using the 'dgetrf' procedure from ACML library to implement computations inside the code fragments.

5 Related Works

An active research intended for increasing the level of programming takes place in different countries as well as in Russia. In Russia, the closest projects are T-system [4] and mpC [5]. From the foreign projects we would like to single out ALF [6] and RapidMind [7].

A large body of associated research is carried out in the field of producing high-performance libraries for linear algebra (ATLAS [8], Plasma [9]), where the blocked algorithms that are friendly to the cash-memory of processors are developed.

6 Conclusion

The fragmented approach to parallel programming is elaborated in this paper. The programming language and the asynchronous programming system Aspect were designed, developed and tested on model tasks. They are intended to solve problems of numerical modeling.

Further research plans aimed at improving the computation model for its subsequent application in supercomputers with distributed memory architecture, at implementing dynamic load balancing on such architectures and at testing the Aspect system on real application tasks.

References

1. Malyshkin, V.E., Valkovskii, V.A.: Synthesis of Parallel Programs and Systems on the Basis of Computational Models. Nauka, Novosibirsk (1988)
2. Malyshkin, V.E.: Fragmented Programming of Library Parallel Numerical Subroutines. In: 7th International Conference on Software Methodologies, Tools and Techniques, vol. 182, pp. 413–423. IOS Press, Roma (2007)
3. Malyshkin, V.E., Sorokin, S.B., Chajuk, K.G.: Fragmentation of Numerical Algorithms for the Parallel Subroutines Library. In: Proceedings of PaCT 2009 (2009)
4. Moskovsky, A., Roganov, V., Abramov, S.: Parallelism granules aggregation with the T-system. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 293–302. Springer, Heidelberg (2007)
5. Lastovetsky, A.L.: Parallel Computing on Heterogeneous Networks. John Wiley & Sons, Chichester (2003)
6. Accelerated Library Framework for Cell Broadband Engine, `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/41838EDB5A15CCCD002573530063D465`
7. McCool, M., Wadleigh, K., Henderson, B., Lin, H.: Performance evaluation of GPUs using the RapidMind development platform. In: ACM/IEEE Conference on Supercomputing, Tampa, Florida, Article No. 181. ACM, New York (2006)
8. Automatically Tuned Linear Algebra Software (ATLAS), `http://math-atlas.sourceforge.net`
9. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Parallel Computing 35(1), 38–53 (2009)

Analyzing Metadata Performance in Distributed File Systems

Christoph Biardzki[1] and Thomas Ludwig[2]

[1] Leibniz-Rechenzentrum (LRZ)
der Bayerischen Akademie der Wissenschaften,
Boltzmannstr. 1, 85748 Garching, Germany
biardzki@lrz.de

[2] German Climate Computing Centre (DKRZ)
Bundesstrasse 55, 20146 Hamburg, Germany
ludwig@dkrz.de

Abstract. The performance of metadata processing in large distributed file systems currently presents larger challenges than scaling of data throughput. The paper presents a novel, distributed benchmark called DMetabench for measuring the performance of metadata operations (e.g. file creation). DMetabench runs in environments with potentially thousands of nodes and allows an assessment of the scalability of metadata operations. Additionally, precise run-time performance data is preserved which allows for a better understanding of performance artifacts. Validation results from production file systems at the Leibniz Supercomputing Centre (LRZ) are provided and discussed. Possible applications of knowledge about metadata performance scaling include the choice of an optimal parallelization strategy for metadata-itensive workload in a specific runtime environment.

Keywords: distributed file system, metadata, benchmark.

1 Introduction

A paradigm shift to massively distributed computing environments has sparked new interest in scalable storage solutions. Although distributed file systems have been used for many years, they only scaled for specific applications, such as file serving in desktop environments, for much of their existence. Only recently did parallel file systems enable the creation of large storage solutions with 100s of Terabytes. However, determining how to cope with millions or billions of files in an efficient manner remains a challenge. This shifts the attention of the research community from scaling data storage capabilities towards scaling the performance of *metadata* – the information needed to organize and find data in a file system.

Early distributed file systems, such as Sun's NFS [1], basically enabled remote access to a local file system on a server. More recent architectures allow for multiple file servers in a common client name space (e.g. AFS [2]) or a separate storage of data and metadata (e.g. CXFS [3] or Lustre [4]). The objective of adding more servers is to increase capacity and performance while maintaining the illusion of a local file system to the application layer.

2 Metadata Semantics

File system semantics define the operational behavior of a file system and include pre-conditions as well as post-conditions. Semantics can describe the visibility of operations (e.g. the point in time when updates are visible in a distributed environment), persistence and durability guarantees or file system invariants (e.g. a file path uniquely describes a file). The behavior of distributed file systems originally comes from local disk file systems which in turn adhere to standards like, for example, POSIX [5]. However, several guarantees which are trivial within a single operating system instance, like the instant visibility of write operations, can be difficult to efficiently implement in a distributed environment. Thus, some distributed file systems have chosen to relax some of the guarantees and to define additional procedures. For example, the *open-to-close* semantics of AFS dictate that writes to a file are only visible to other processes after a *close()* or an explicit flush operation. Other semantics like *close-to-open* or *immutable* are also being used. Such relaxations can reduce the amount of locking and thus improve performance.

Remarkably, some metadata-focused operations, such as creating a new file, still carry the uniqueness guarantees given by a local file system and therefore prohibit any shortcuts regarding access coordination in a distributed environment. In this way, the very basic process of creating a file is a potential bottleneck in distributed file systems depending on the locking mechanisms involved. Performance of non-modifying metadata operations, such as reading file attributes, can also suffer from network latency depending on the presence of caches. Some real-life workloads, for example filesystem-based mail servers, search engines, data backup, replication or virus scanning are inherently metadata-intensive.

To sidestep latency, a parallelization of operations (e.g. creating multiple files concurrently) is a possible solution. The attractivity of this approach is further enhanced by the fact that distributed file systems often include large disk arrays which are well-suited to processing multiple requests at the same time by request distribution. From the client perspective the current architectural trend towards multi-core architectures enables increased concurrency inside a single operating system (OS) instance and thus also more simultaneous – but potentially conflicting – metadata operations (*intra-node* concurrency). On the other hand some distributed file systems were optimized for *inter-node* concurrency, which means access from *multiple different* OS instances. These two types have profound implications on the performance of concurrent metadata operations because often different consistency mechanisms are used. Furthermore most distributed file systems allow to tune parameters which influence concurrency limits. Thus it is quite difficult to find out how to parallelize a metadata-intensive task in a way which achieves optimum performance.

3 DMetabench - A Distributed Metadata Benchmark

To gain more insight into the topic of parallelizing metadata operations a special metadata microbenchmark called DMetabench has been developed at LRZ. DMetabench is implemented in Python and MPI and uses a master-worker architecture (fig. 1) to start multiple parallel processes and then perform selected metadata operations on a common distributed file system. Typical MPI runtime environments for DMetabench include a wide spectrum of distributed systems from single nodes to large compute clusters and also large SMP machines.

A metadata operation (e.g. 'delete file') is defined as a plug-in written in Python and using standard file system APIs such as *open()*. DMetabench then allows to estimate how many of these operations per time unit can be performed using a given number of processes and a given file system. Every benchmark run consists of a preparation, a benchmark and a cleanup phase. This is useful to fulfill preconditions (e.g. files must exist before deletion). An example of a complete benchmark plugin is *MakeFiles* which creates empty files in a separate directory for every process and thus stresses the ability of the filesystem to generate new file metadata.

One distinctive feature of DMetabench is the ability to recognize the placement of processes within the OS instance boundaries in the MPI environment. Using this technique the intra- and inter-node cases can be differentiated and DMetabench automatically iterates over possible combinations of the number of processes per OS instance and number of OS instances (fig. 2). The resulting data helps to pinpoint the optimal parallelization style for a given environment and gathers data for different setups in a single benchmark run.

Additionally, the runtime performance of by each process is logged in fine-granular intervals (approximately 0.1s) to capture information about variations in performance. In comparison, existing benchmarks often only measure the runtime and the number of operations to calculate an "average" performance and lose much interesting information in this way. The usefulness of the additional data will be shown later in section 4.

To facilitate the evaluation of results measurement data from DMetabench can be easily visualized using the two-step process shown in fig. 3. First, the time-stamped performance data is aggregated to obtain summary data, e.g. for all processes on a node. Then the operator can decide to compare multiple data sets in a single diagram using auxiliary scripts which help to create the necessary control files for the plotting software. There are three different types of diagrams available: the *runtime chart* (fig. 4) shows the summary performance for the participating processes during a measurement. The upper part shows the number of operations completed. In the mid part, an indicator called coefficient of variation (COV) shows the ratio of standard deviation of single-process performance to the mean value. The COV helps to assess whether there are large differences in the speed of particular processes. Large values can indicate problems with the test setup (e.g. network problems for a single compute node). The bottom part shows the momentary aggregate performance for all processes.

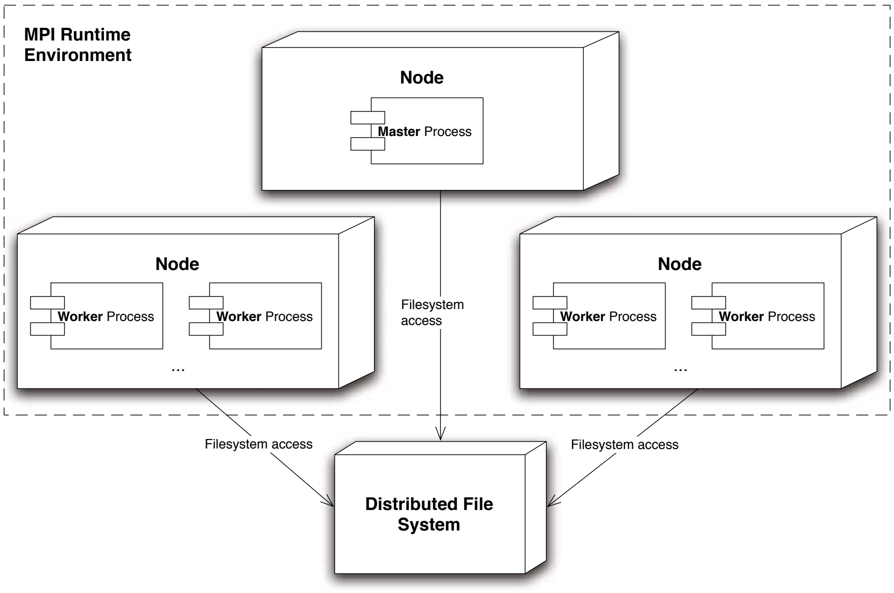


Fig. 1. Deployment diagram for DMetabench

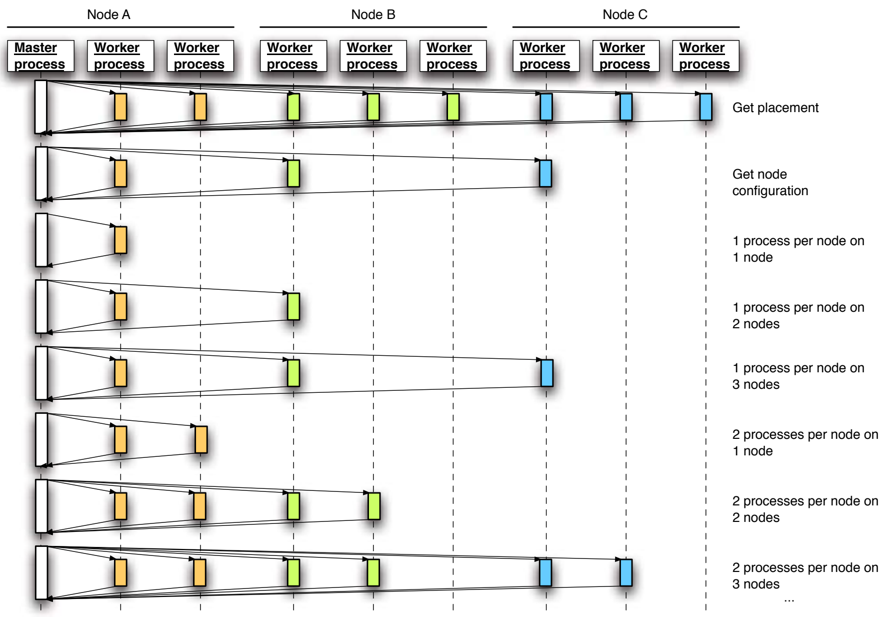

Node A
Node B
Node C
Master process
Worker process
Get placement
Get node configuration
1 process per node on 1 node
1 process per node on 2 nodes
1 process per node on 3 nodes
2 processes per node on 1 node
2 processes per node on 2 nodes
2 processes per node on 3 nodes
...

Fig. 2. Example intra-/internode measurement sequence for nine processes on three nodes

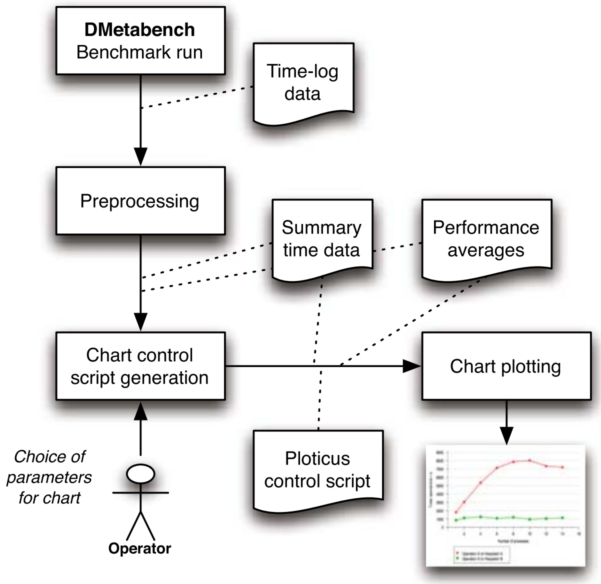


Fig. 3. DMetabench workflow for preprocessing and comparing data using charts

The two other charts (fig. 5 and fig. 6) visualize the scaling of a particular operation type for a given number of processes respectively a given number of nodes (operating system instances). The specifics of the example figure will be discussed in detail later.

4 Measurements on Production Systems

DMetabench has been used on a variety of distributed file systems at the Leibniz Supercomputing Centre (LRZ) in Munich. LRZ hosts one of Germany's national supercomputers, the HLRB2, which is a supercluster with nineteen 512-core SGI Altix 4700 systems. The HLRB2 uses a 600 TB SGI CXFS file system for temporary data and a clustered NFS-based file server (Ontap GX [6]) for home and project data. Additionally a 600+ node linux cluster at LRZ uses both NFS-based file servers and also a Lustre parallel file system. Three selected examples shown below were chosen mainly for their qualitative characteristics and not because of absolute performance numbers which of course depend on the particular hardware setup and software version.

The chart in fig. 4 illustrates how the time-logging feature of DMetabench helps to understand the runtime environment. It shows twenty compute nodes with one process per node which create files in parallel on a shared NFS file system. Line (a) shows an undisturbed measurement and in (b) a CPU-hogging process was started on one of the nodes during t=19s to t=28s. The middle COV graph clearly shows an elevated plateau indicating a larger difference in the performance of the particular processes. A conventional benchmark, which averages

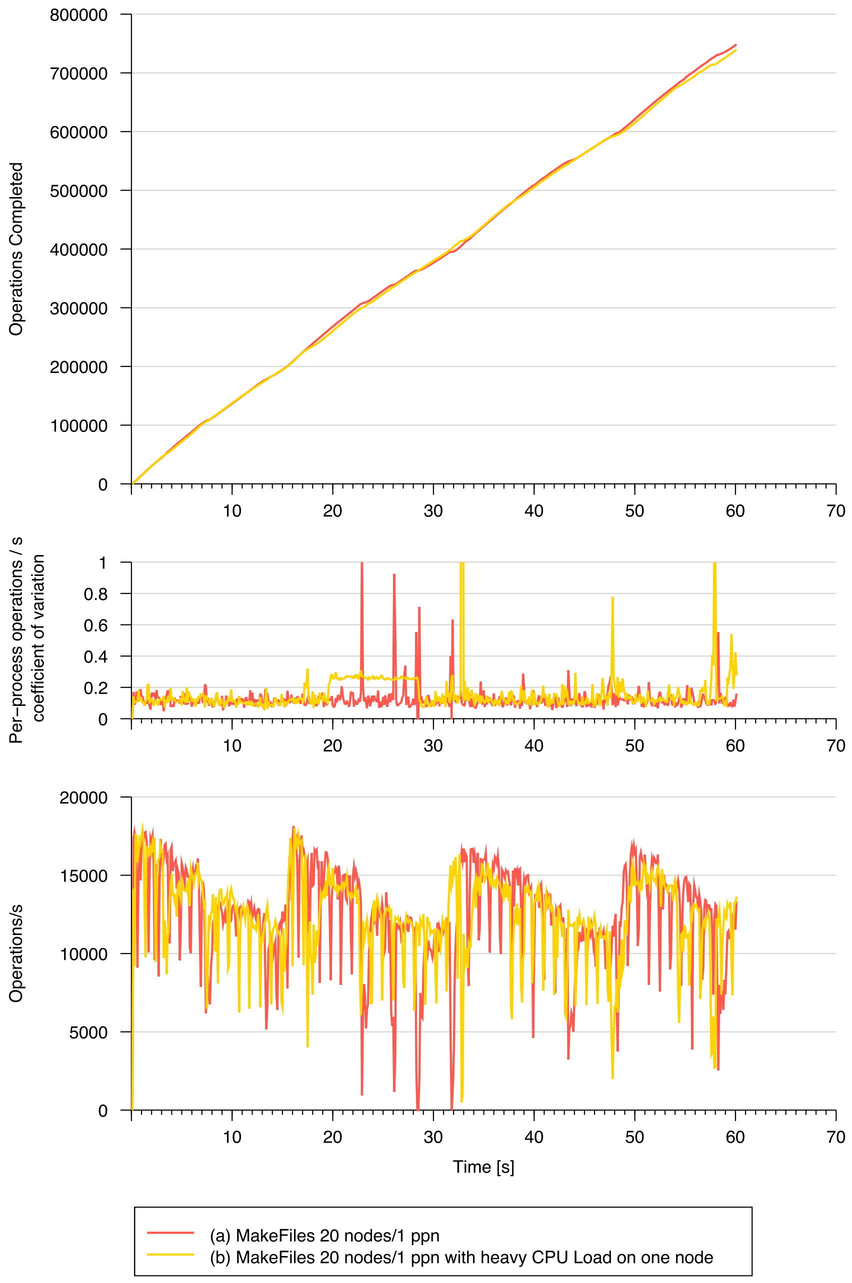


Fig. 4. Example runtime performance chart generated with DMetabench

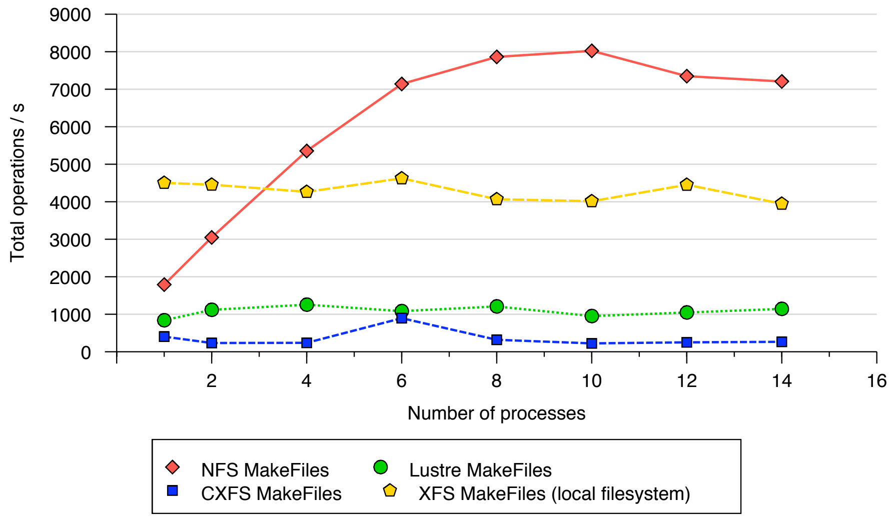


Fig. 5. Comparison of multi-process scaling of three different remote file systems (NFS, CXFS and Lustre) and a local XFS file system on a single node

runtime and the number of operations completed, would have missed such artifacts. On the other hand the bottom graph with the total performance exhibits a regular sawtooth pattern, which means that total file creation performance varies during the measurement. The reason here is the WAFL[7] file system used by the measured NFS file server which performs regular consistency points. This process takes away processing time, so that file creation performance is reduced. Here, DMetabench allows to observe the inner working of a file server from a black-box perspective. Very similar results can be obtained with Lustre and the underlying ext3 file system with its regular 10s buffer cache flush intervals.

Figure 5 demonstrates 0-byte file creation characteristics of different file systems on the same 16-core SMP machine with up to 14 processes. A local XFS [8] file system has an almost equal performance with both a single and many processes. This explains why there were hardly any efforts to parallelize metadata-intensive workloads – it does not really matter for *local* file systems. On the contrary NFS begins with a much slower single process performance but then scales up to ten processes. Both Lustre and CXFS are much slower and do not scale up with more processes.

Finally figure 6 and 7 demonstrate intra- and inter-node scaling of two different file systems (NFS and Lustre). NFS scales both within a single compute node and with multiple nodes up to the performance limit of the NFS file server. For example the performance of 4 processes on one node, 4 processes on two nodes (2 per node) and 4 processes on four nodes (1 per node) are quite similar. Lustre (in version 1.6) does not scale within a single OS instance which can be a problem for large SMP nodes. The reason is that there is a hard-coded limit of one simultaneous modifying metadata operation per node in Lustre while

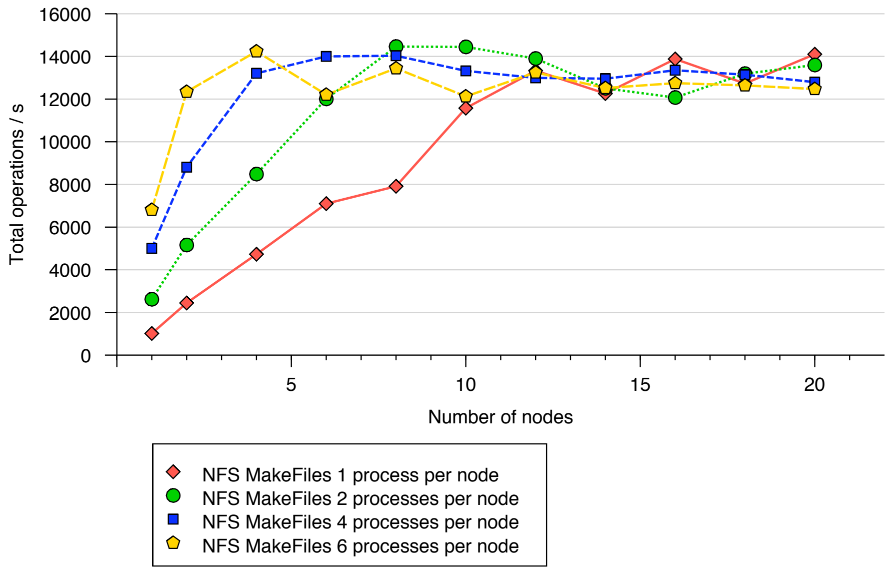


Fig. 6. Multi-node and multi-process file creation with NFS

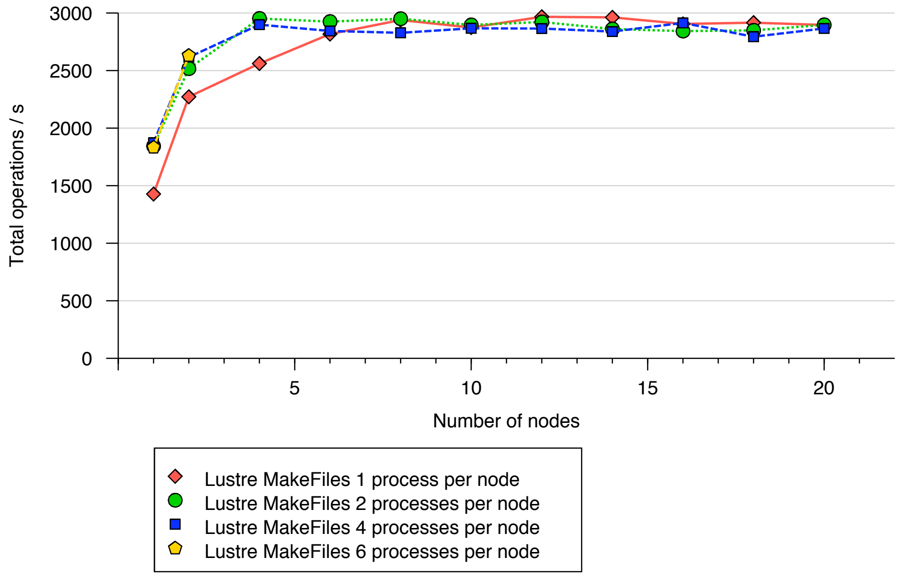


Fig. 7. Multi-node and multi-process file creation on Lustre

non-modifiying operations like *stat()* can be issued concurrently. NFS on Linux has higher limits for simultaneous operations which helps to achieve better intra-node scaling. This kind of differences has direct practical applications on the

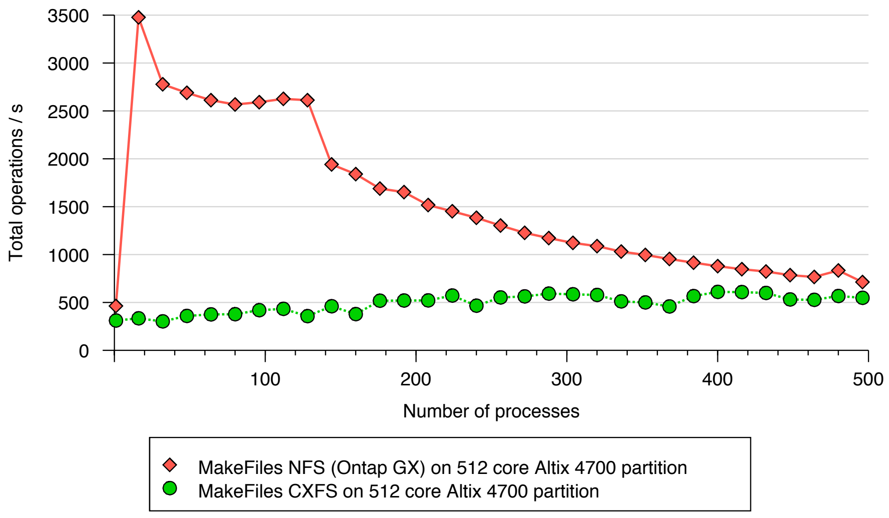


Fig. 8. File creation performance of NFS (Netapp Ontap GX) and a SAN file system (SGI CXFS)

choice of file systems for specific applications. For example a maildir-format based mail server like Postfix exhibits a high percentage of concurrent file creations during mail delivery and requires good intra-node scaling. On the other hand Lustre can handle file creations from thousands of nodes (inter-node) even if its performance within a single node is limited.

Early versions of DMetabench have been used successfully at LRZ for benchmarking and acceptance tests of file systems during procurements of large storage systems for supercomputers (e.g. the HLRB2). Among others extensive testing of a clustered NAS system (Ontap GX) has been performed. As an example, fig. 8 shows the difference in file creation performance between the NAS system and a shared SAN-File system on a 512-core Altix 4700 SMP system.

5 Related Work

File system benchmarking has been traditionally used to evaluate improvements in existing file systems as well as for testing new architectures. There are many I/O benchmarks which focus on data throughput, e.g. IOzone [9], IOmeter [10] or b_eff_io [11]. The typical workload includes different types of access to the data in a file with very little metadata activity.

Well-known benchmarks with a focus on metadata handling include the Andrew Benchmark [12], Netapp's single-threaded Postmark [13] or the official SPEC SFS 97 for NFS and SFS 2008 [14] for NFS and CIFS. The Andrew Benchmark and Postmark are application-level benchmarks which use the OS API to access a file system. In contrast, SPEC SFS is filesystem-specific and generates NFS and CIFS-protocol packets and bypasses the operating system. All these benchmarks intermix data and metadata workloads to resemble real-life workloads while DMetabench tries to explicitly avoid data access and thus establishs an upper limit of performance for pure metadata operations. An interesting, recent metadata benchmark is Filebench [15] but it does not currently support distributed execution.

6 Summary

DMetabench is a microbenchmark which can be used to qualitatively measure the performance of concurrent metadata operations in distributed file systems. It is specifically useful to optimize the concurrency and placement of metadata-intensive workloads when both multiple nodes and multiple processes per node are possible options. DMetabench runs on standard MPI environments and has been tested with hundreds of processes both on clusters and large SMP systems. Besides showing scaling properties of a file system the fine-grained time-logging feature of DMetabench allows observation of inner workings of a file system from a black-box perspective.

References

1. Callaghan, B., Pawlowski, B., Staubach, P.: NFS Version 3 Protocol Specification (1995), http://www.ietf.org/rfc/rfc1813.txt
2. Campbell, R.: Managing AFS: The Andrew File System. Prentice-Hall, Englewood Cliffs (1998)
3. Shepard, L., Eppe, E.: SGI InfiniteStorage Shares Filesystem CXFS: A High-Performance, Multi-OS Filesystem from SGI. Technical report, Silicon Graphics (2006)
4. Cluster File Systems, Inc.: Lustre 1.6 Operations Manual (2007)
5. The Open Group: The Single UNIX Specification, Version 3. Technical report (2004)
6. Eisler, M., Corbett, P., Kazar, M., Nydick, D.S., Wagner, J.C.: Data ONTAP GX: A Scalable Storage Cluster. In: Proceedings of FAST 2007 (2007)
7. Hitz, D., Lau, J., Malcolm, M.: File System Design for an NFS File Server Appliance. Technical report, Network Appliance (TR 3002)
8. Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., Peck, G.: Scalability in the XFS File System. In: Proceedings of the USENIX 1996 Technical Conference, San Diego, CA, USA, pp. 1–14 (22–26 1996)
9. Norcott, W.D., Capps, D.: Iozone Filesystem Benchmark (2006), http://www.iozone.org/
10. Intel Corporation: Iometer (1998), http://www.iometer.org/
11. Rabenseifner, R., Koniges, A.E., Prost, J.P., Hedges, R.: The Parallel Effective I/O Bandwidth Benchmark: b_eff_io. Technical report, High-Performance Computing Center, HLRS (2001)

12. Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M.: Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems 6, 51–81 (1988)
13. Katcher, J.: PostMark: A new file system benchmark. Technical report 3022, Network Appliance (1997)
14. SPEC: SPECsfs 2008 User's Guide. Technical Report Version 1.0, Standard Performance Evaluation Corporation (SPEC) (2008)
15. McDougall, R., Mauro, J.: Filebench tutorial (2006), http://www.solarisinternals.com/si/tools/filebench

Towards Parametric Verification of Prioritized Time Petri Nets*

Anna Dedova[1] and Irina Virbitskaite[1,2]

[1] Novosibirsk State University
2, Pirogova st., Novosibirsk, 630090, Russia
[2] A.P. Ershov Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentiev avenue, 630090, Novosibirsk, Russia
annavd@ngs.ru, virb@iis.nsk.su

Abstract. The intention of the paper is to develop an algorithm for parametric timing behaviour verification of real-time and concurrent systems represented by prioritized time Petri nets (PrTPNs). To achieve the purpose, we introduce a notion of the parametric PrTPN which is a modification of the PrTPN by using parameter variables in specification of timing constraints on transition firings. System properties are given as formulae of a parametric extension of the real-time branching time temporal logic TCTL, PTCTL. The verification algorithm consists in constructing conditions on timing parameter variables under which the PrTPN with bounded parameters works w.r.t. the checked PTCTL-formula. It is also shown the correctness and evaluated the complexity of the algorithm proposed.

1 Introduction

Within the last two decades, serious attempts have been made to extend the success of model checking to the real-time setting — timed models (e.g., timed automata [1], time Petri nets [8]) and timed temporal logics (see, for example, [1]). One of the major obstacles for real-time model checking is that it usually requires overly detailed specification of timing characteristics of both the system and its properties. In the case when the checked formula is not satisfied by the system the timing characteristics are changed, and verification algorithm is applied again. It leaves users in repetitive trial-and-error cycles to select proper timing characteristics. One of the ways out is parametric reasoning working on a model with parameters — symbolic constants with unknown, fixed values. In [3], Alur et al. have introduced parameters in discrete- and dense-timed automata and have shown that the emptiness problem is decidable when only one clock is compared to parameters. In [2,6,12], the authors have introduced parameters in temporal logics and established that the model-checking problem for TCTL

* This work is supported in part by the DFG-RFBR (grant No 436 RUS 113/1002/01,09-01-91334).

extended with parameters (PTCTL) over discrete- and dense-timed automata (without parameters) is decidable. The paper [7] has studied the model-checking and parameter synthesis problems of the language PTCTL over discrete-timed automata with parameters. It has turned out that the problems are undecidable over discrete-timed automata with only one parametric clock. In [6,7] the durations of runs of a timed automata are shown to be expressible in the arithmetic of Presburger (when the time domain is discrete) and the theory of the reals (when the time domain is dense). The paper [11] provided a timing behaviour analysis algorithm for one-safe time Petri nets and TCTL-formulae using cactus structures [12] to calculate the durations of runs. More recently, "on-the-fly" model checking algorithms for parametric time Petri nets with stopwatches w.r.t. a subset of PTCTL-formulae have been put forward in [9].

The intention of the paper is to develop an algorithm for parametric timing behaviour verification of real-time and concurrent systems represented by prioritized time Petri nets (PrTPNs). To fulfill the purpose, we introduce a notion of the parametric PrTPN which is a modification of the PrTPN by using parameter variables in specification of timing constraints on transition firings. Net properties are given as formulae of PTCTL. The durations of computational paths are expressed in formulae of the real arithmetic [6]. A timing behaviour analysis algorithm consists in constructing conditions on timing parameter variables under which the PrTPN with bounded parameters works w.r.t. the checked PTCTL-formula.

2 Parametric Prioritized Time Petri Nets

In this section, we fist define some notations which are needed to introduce parameters into the net model and logic formulae, and then consider some terminology concerning parametric prioritized time Petri nets.

Let $\mathbf{N}$ be the set of natural numbers. Also, let $\mathbf{R}$ be the set of nonnegative real numbers and $\mathbf{R}^+$ the set of positive real numbers. Assume a finite set Θ of parameters θ that are shared by the net model and the logical formulae. Let θ with and without subscripts range over Θ. A *parameter valuation* χ for Θ is a mapping from Θ into $\mathbf{N}$ which assigns a natural number to each parameter θ from Θ. From now on, α, β, ... mean any linear term $\sum\limits_{j\in J} c_j\theta_j + c$ with $c_j, c \in \mathbf{N}$ and $J \subset \{1, ..., n\}$. A parameter valuation χ can be naturally extended to linear terms by defining $\chi(c) = c$ for any $c \in \mathbf{N}$. We shall use $\mathcal{T}$ to denote the set of linear terms. Let $\mathcal{I}(\mathbf{N}, \mathcal{T})$ be the set of parametric time intervals i such that the left end-point of i, $\downarrow i$, belongs to $\mathbf{N} \cup \mathcal{T}$ and the right end-point of i, $\uparrow i$, belongs to $\mathbf{N} \cup \mathcal{T} \cup \{\infty\}$. Given $i \in \mathcal{I}(\mathbf{N}, \mathcal{T})$ and a parameter valuation χ, i^χ denotes the time interval obtained from i by replacing every occurrence of parameters θ with $\chi(\theta)$.

We now consider the notion of Petri nets. A *Petri net* is a tuple $N = (P, T, {}^\bullet(\cdot), (\cdot)^\bullet, m^*)$, where P is a finite set of places, T is a finite set of transitions ($P \cap T = \emptyset$), ${}^\bullet(\cdot) \in (\mathbf{N}^P)^T$ (resp. $(\cdot)^\bullet \in (\mathbf{N}^P)^T$) is the forward (resp. backward) incidence mapping, $m^* \in \mathbf{N}^P$ is the initial marking. We shall use ${}^\bullet t$ (resp. $t^\bullet$) to denote the set of places ${}^{\bullet}t = \{p \in P \mid {}^{\bullet}t(p) > 0\}$ (resp. $t^{\bullet} = \{p \in P \mid t^{\bullet}(p) > 0\}$). A *marking* m of N is a mapping from $\mathbf{N}^P$. A transition t is *enabled* in a marking m if $m \geq^{\bullet} t$, otherwise it is *disabled*. Let $enable(m)$ be the set of transitions, enabled in m. Define a predicate $\uparrow enabled(t', m, t) \in \{true, false\}$ which is true, if a transition t' is newly enabled after firing a transition t in a marking m, and false, otherwise: $\uparrow enabled(t', m, t) = [t' \in enabled(m -^{\bullet} t + t^{\bullet})] \wedge [t' \notin enabled(m -^{\bullet} t) \vee (t = t')]$.

Time Petri Nets were introduced in [8] and extend Petri Nets with timing constraints on the firings of transitions. An extension of time Petri nets with priorities (PrTPNs for short) has been proposed in [5]. In a PrTPN, a transition is not allowed to fire if some transition with higher priority is fireable at the same instant. We introduce an extension of PrTPNs — parametric PrTPNs whose transitions are associated with time predicates representing unspecified timing constraints on transition firings. Let $\mathcal{V} = [T \rightarrow \mathbf{R}]$ be the set of time assignments for transitions from T.

Definition 1. *A* parametric prioritized time Petri net (PPrTPN) *is a tuple* $\mathcal{N} = (P, T, {}^{\bullet}(\cdot), (\cdot)^{\bullet}, m^*, \succ, \Theta, I, \nu^*)$*, where* $(P, T, {}^{\bullet}(\cdot), (\cdot)^{\bullet}, m^*)$ *is a Petri net,* $\succ \in T \times T$ *is a transitive, asymmetric, irreflexive binary* priority *relation,* Θ *is a finite set of parameters* $(\Theta \cap (P \cup T) = \emptyset)$, $I : T \rightarrow \mathcal{I}(\mathbf{N}, \mathcal{T})$ *is a function that associates each transition t with a parametric time interval* $I(t) \in \mathcal{I}(\mathbf{N}, \mathcal{T})$, $\nu^* \in \mathcal{V}$ *is the* initial time assignment. *Let* $\Theta_{\mathcal{N}}$ *to denote the set of parameters appearing in linear terms in a specification of* $\mathcal{N}$*.*

The semantics of a PPrTPN $\mathcal{N}$ is defined at a parameter valuation χ. From now on, $\mathcal{N}^{\chi}$ means a PrTPN obtained from PPrTPN $\mathcal{N}$ by replacing every occurrence of a parameter θ with $\chi(\theta)$ for all $\theta \in \Theta_{\mathcal{N}}$. A *state* q of $\mathcal{N}^{\chi}$ is a pair $\langle m, \nu \rangle$, where m is a marking of $\mathcal{N}^{\chi}$ and $\nu \in \mathcal{V}$. The *initial state* of $\mathcal{N}^{\chi}$ is the pair $q^* = \langle m^*, \nu^* \rangle$. The states of $\mathcal{N}^{\chi}$ change, if time passes or if a transition fires. Let $q = \langle m, \nu \rangle, q' = \langle m', \nu' \rangle$ be states of $\mathcal{N}^{\chi}$. In a state q, time $\delta \in \mathbf{R}^+$ *can pass*, if for all $t \in enable(m)$ there exists $\delta' \geq \delta$ such that $\nu(t) + \delta' \in (I(t))^{\chi}$. In this case, the state q' is *obtained by passing* δ from q (written $q \overset{\delta}{\Rightarrow} q'$), if $m' = m$ and $\nu' = \nu + \delta$. In a state q, a transition $t \in T$ is *fireable*, if $t \in enable(m)$, $\nu(t) \in (I(t))^{\chi}$, and for all $t' \in enabled(m)$ if $t' \succ t$ then $\nu(t') \notin (I(t))^{\chi}$. In this case, the state q' is *obtained by firing* t from q (written $q \overset{0}{\Rightarrow} q'$ or $q \overset{t}{\Rightarrow} q'$), if $m' = m -^{\bullet} t + t^{\bullet}$, and $\forall t' \in T \diamond \nu'(t') = \begin{cases} 0, & \text{if } \uparrow enabled(t', m, t), \\ \nu(t'), & \text{otherwise.} \end{cases}$ A *q-run* (*run*) r of $\mathcal{N}^{\chi}$ is a finite (infinite) sequence $r = (q_i)_{0 \leq i \leq j}$ $(r = (q_i)_{i \geq 0})$ of states and real numbers $\delta_i \in \mathbf{R}$ of the form: $q = q_0 \overset{\delta_0}{\Rightarrow} q_1 \ldots q_{j-1} \overset{\delta_{j-1}}{\Rightarrow} q_j$ $(q = q_0 \overset{\delta_0}{\Rightarrow} q_1 \ldots q_{n-1} \overset{\delta_{n-1}}{\Rightarrow} q_n \ldots)$. A *position* $\mathbf{p}$ in r is a state $q_i + \delta$, where either $\delta = 0$ or $0 < \delta < \delta_i$ $(0 \leq i < j$ or $i \geq 0)$. The *duration* $D(r, \mathbf{p})$ of a run r in a position $\mathbf{p} = q_j + \delta$ is equal to $\sum_{0 \leq i < j} \delta_i + \delta$. The set of positions in a run r can be totally ordered as follows. Let $\mathbf{p} = q_i + \delta$ and $\mathbf{p}' = q_{i'} + \delta'$ be two positions in r. Then $\mathbf{p} < \mathbf{p}'$ iff either $i < i'$ or $i = i'$ and $\delta < \delta'$. We shall write $\mathbf{p} \leq \mathbf{p}'$ if $\mathbf{p} < \mathbf{p}'$ or $\mathbf{p} = \mathbf{p}'$. A state q is *reachable* in $\mathcal{N}^{\chi}$ if it appears in a q^*-run of $\mathcal{N}^\chi$. Let $RS(\mathcal{N}^\chi)$ denote the set of all reachable states of $\mathcal{N}^\chi$. To guarantee that in any run of $\mathcal{N}^\chi$ time is increasing beyond any bound, we need the following *progress condition*: for every set of transitions $\{t_1, t_2, \ldots, t_n\}$ s.t. $\forall\, 1 \leq i < n \diamond t_i^\bullet \cap {}^\bullet t_{i+1} \neq \emptyset$ and $t_n^\bullet \cap {}^\bullet t_1 \neq \emptyset$ it holds $\sum_{1 \leq i \leq n} \downarrow (I(t_i))^\chi > 0$. We call $\mathcal{N}^\chi$ *bounded*, if there is $K \in \mathbf{N}$ such that for any $\langle m, \nu \rangle \in RS(\mathcal{N}^\chi)$ and any $p \in P$ holds $m(p) \leq K$. In the sequel, $\mathcal{N}^\chi$ will always denote a bounded PrTPN satisfying the progress condition.

3 PTCTL: Syntax and Semantics

In this section, we review the syntax and semantics of PTCTL (Parametric Timed Computation Tree Logic) proposed in [12].

Definition 2. *The* PTCTL-formula φ *is is inductively defined by the following grammar:* $\phi ::= \mathcal{P} \mid \neg\phi \mid \phi \vee \phi \mid \alpha \sim \beta \mid \phi\ Q\ U_{\sim\alpha}\phi$, *where* $\sim \in \{<, \leq, =, \geq, >\}$, $Q \in \{\exists, \forall\}$, $\mathcal{P} \in PR$ *and* $PR = \{\mathcal{P} \mid \mathcal{P} : m \rightarrow \{true, false\}\}$ *is a set of propositions on the net marking. The set of free parameters of* φ *is denoted by* Θ_φ.

Given a PTCTL-formula φ and a parameter valuation χ, we let φ^χ be the PTCTL-formula obtained from φ by replacing every occurrence of θ with $\chi(\theta)$ for all $\theta \in \Theta_\varphi$. PTCTL-formulae φ^χ are interpreted on the states of a model $\mathcal{M} = (RS(\mathcal{N}^\chi), \mathcal{W})$, where $\mathcal{W} : RS(\mathcal{N}^\chi) \rightarrow 2^{PR}$ is a function such that $\mathcal{W}(q = \langle m, \nu \rangle) = \{\mathcal{P} \in PR \mid \mathcal{P}(m) = true\}$. Given a state $q \in RS(\mathcal{N}^\chi)$ and a PTCTL-formula φ^χ, the *satisfaction* relation $\mathcal{N}^\chi, q \models \varphi^\chi$ is defined inductively as follows:

$\mathcal{N}^\chi, q \models \mathcal{P}^\chi \iff \mathcal{P} \in \mathcal{W}(q)$
$\mathcal{N}^\chi, q \models (\neg\phi)^\chi \iff \mathcal{N}^\chi, q \not\models \phi^\chi$
$\mathcal{N}^\chi, q \models (\phi \vee \psi)^\chi \iff \mathcal{N}^\chi, q \models \phi^\chi$ or $\mathcal{N}^\chi, q \models \psi^\chi$
$\mathcal{N}^\chi, q \models (\alpha \sim \beta)^\chi \iff \chi(\alpha) \sim \chi(\beta)$
$\mathcal{N}^\chi, q \models (\phi\ Q\ U_{\sim\alpha}\psi)^\chi \iff$ for any/some (depending on Q) q-run $r = (q_i)_{i \geq 0}$ in $\mathcal{N}^\chi$, there exists a position $\mathbf{p}$ in r such that $D(r, \mathbf{p}) \sim \chi(\alpha)$, $\mathcal{N}^\chi, \mathbf{p} \models \psi^\chi$ and $\mathcal{N}^\chi, \mathbf{p}' \models \phi^\chi$ for all positions $\mathbf{p}'$ in r such that $\mathbf{p}' < \mathbf{p}$

We say that $\mathcal{N}^\chi$ satisfies φ^χ (written $\mathcal{N}^\chi \models \varphi^\chi$) iff $\mathcal{N}^\chi, q^* \models \varphi^\chi$.

The parametric timing behaviour analysis problem $\mathcal{PTBA}(\mathcal{N}, \varphi)$ is formulated as follows: compute a symbolic representation of the set of parameter valuations χ on Θ_φ such that $\mathcal{N}^\chi \models \varphi^\chi$. The structural translation preserving timed language acceptance proposed in [4] from a TA into a bounded TPN can straightforwardly be extended to parametric TA. As the emptiness problem (and then, the reachability problem) is undecidable for parametric TA [3], it is also undecidable for parametric TPNs. Since the emptiness problem is a particular case of the model checking problem, the latter is undecidable for parametric TPNs and hence for parametric PrTPNs. Thus, $\mathcal{PTBA}(\mathcal{N}, \varphi)$ is undecidable because it is a more general problem.

4 Parametric Timing Behaviour Analysis

First, we recall the definition of regions (equivalence classes of states) and region graphs [1] in order to get a finite representation of the state-space of the PrTPN $\mathcal{N}^\chi$. Let $c_{\mathcal{N}^\chi}$ mean the biggest constant from $\mathbf{N}$ appearing as the endpoint of a time interval in $\mathcal{N}^\chi$. For any $\delta \in \mathbf{R}$, $\{\delta\}$ denotes the fractional part of δ, and $\lfloor \delta \rfloor$ denotes the integral part of δ. Given $\nu, \nu' \in \mathcal{V}$, $\nu \simeq \nu'$ iff the following conditions are met: (i) for each $t \in T$: either $\lfloor \nu(t) \rfloor = \lfloor \nu'(t) \rfloor$ or $\nu(t), \nu'(t) > c_{\mathcal{N}^\chi}$, (ii) for each $t, t' \in T$ such that $\nu(t) \leq c_{\mathcal{N}^\chi}$ and $\nu'(t) \leq c_{\mathcal{N}^\chi}$: (a) $\{\nu(t)\} \leq \{\nu(t')\} \Leftrightarrow \{\nu'(t)\} \leq \{\nu'(t')\}$; (b) $\{\nu(t)\} = 0 \Leftrightarrow \{\nu'(t)\} = 0$. Given $\nu \in \mathcal{V}$, we use $[\nu]$ to denote the equivalence class of ν w.r.t. $\simeq$. A *region* of $\mathcal{N}^\chi$ is called to be a set $[q] = \langle m, [\nu] \rangle = \{\langle m', \nu' \rangle \in RS(\mathcal{N}^\chi) \mid m = m' \wedge \nu' \simeq \nu\}$. A region $\langle m, [\nu] \rangle$ is called *boundary*, if $\nu \not\simeq \nu + \delta$ for any $\delta > 0$; *unbounded*, if $\nu(t) > c_{\mathcal{N}^\chi}$ for any $t \in T$. A predicate $B(v)$ is true, if v is boundary region, and false, otherwise. For $\langle m, [\nu] \rangle \neq \langle m', [\nu'] \rangle$, $\langle m', [\nu'] \rangle$ is said to be a *successor* of $\langle m, [\nu] \rangle$ (written $\langle m', [\nu'] \rangle = succ(\langle m, [\nu] \rangle))$, if $m = m'$, $\nu' = \nu + \delta$ for some positive $\delta \in \mathbf{R}^+$ and $\nu + \delta' \in [\nu] \cup [\nu']$ for all $\delta' < \delta$. The *region graph* of $\mathcal{N}^\chi$ is defined to be the labelled directed graph $G(\mathcal{N}^\chi) = (V, E, l)$. The vertex set V is the set of all regions of $\mathcal{N}^\chi$. The edge set E consists of two types of edges: (i) the edge $(\langle m, [\nu] \rangle, \langle m', [\nu'] \rangle)$ may represent firing a transition if $\langle m', \nu' \rangle$ is obtained from $\langle m, \nu \rangle$ by firing some $t \in T$; (ii) the edge $(\langle m, [\nu] \rangle, \langle m', [\nu'] \rangle)$ may represent the passage of time if either $\langle m', [\nu'] \rangle = succ(\langle m, [\nu] \rangle)$ or $\langle m, [\nu] \rangle = \langle m', [\nu'] \rangle$ is an unbounded region. The function l labels an edge either with the symbol $'t'$ (if the edge represents firing t) or with the symbol $'\delta'$ (if the edge represents the passage of time). It is well-known that the size of the region graph $G(\mathcal{N}^\chi)$ is bounded by $2^{|\mathcal{N}^\chi|}$. From now on, v^* denotes the initial region $[q^*]$ of $G(\mathcal{N}^\chi)$. There is a correspondence between runs r in $\mathcal{N}^\chi$ and paths ρ in $G(\mathcal{N}^\chi)$. Let $r = (q_i)_{i \geq 0}$. Consider $q_i \overset{\delta_i}{\Rightarrow} q_{i+1}$. If $\delta_i = 0$ or $[q_i] = [q_{i+1}]$ is an unbounded region, then $([q_i], [q_{i+1}])$ is an edge in $G(\mathcal{N}^\chi)$, according to the definition of $G(\mathcal{N}^\chi)$. If $\delta_i > 0$, then there are positions p_j $(0 \leq j \leq n_i + 1)$ in r such that $q_i = p_0$, $q_{i+1} = p_{n_i+1}$, and $[p_{j+1}] = succ([p_j])$ for all $0 \leq j \leq n_i$. In this case, the obtained path $\pi(r)$ in $G(\mathcal{N}^\chi)$ corresponds to the run r in $\mathcal{N}^\chi$, and we say that $\pi(r)$ is the path *associated* with r.

Next, we use the theory of the reals to calculate run durations. Real Arithmetic (RA) is the set of first-order formulae of $\langle \mathbf{R}, +, <, N, 0, 1 \rangle$, where N is a unary predicate. The RA-formulae are interpreted over the real numbers. The interpretation of N is defined such that $N(x)$ holds iff x is a natural number. RA has a decidable theory with complexity in 3ExpTime in the size of the sentence [6]. Consider the definitions of auxiliary sets. Given a region graph $G^\chi = (V, E, l)$ with $v, v' \in V$ and $S \subseteq V$, we define $\lambda^\chi_{S,v,v'}$ as the set of $x \in \mathbf{R}$ such that (i) there exists a finite run $r = (q_i)_{0 \leq i \leq j}$ in $\mathcal{N}^\chi$ with duration $x = D(r, q_j)$, (ii) $v = v_0, v' = v_k$ and $v_l \in S$ $(0 \leq l < k)$ for the path $\pi(r) = (v_l)_{0 \leq l \leq k}$ in G^χ, associated with r. Let $\phi(y)$ be a formula with a single free variable y. A set $Y \subseteq \mathbf{N}$ is *definable* by an RA-formula if Y is the set of all assignments of variable y making the formula $\phi(y)$ true.

Proposition 1. *Given a region graph $G^\chi = (V, E, l)$ with $v, v' \in V$ and $S \subseteq V$, the set $\lambda^\chi_{S,v,v'}$ is definable by an RA-formula; the construction of the formula is effective.*

Finally, we formulate and solve a restricted variant of $\mathcal{PTBA}(\mathcal{N}, \varphi)$. Let $\Omega \subseteq \mathbf{R}^{\Theta_\mathcal{N}}$ be a convex polyhedron that is the domain of the parameters from $\Theta_\mathcal{N}$, and $\Omega_\mathbf{N}$ be the set of the natural valued points of Ω, that is finite and can be defined by using standard techniques. We restrict ourselves to constructing a symbolic representation of parameter valuations on Θ_φ, which belong to $\Omega_\mathbf{N}$ on $\Theta_\mathcal{N}$, and denote the restricted problem as $\mathcal{PTBA}(\mathcal{N}_\Omega, \varphi)$. Define an equivalence relation $\approx$ on the set $\Upsilon = \{\chi \mid \chi|_{\Theta_\mathcal{N}} \in \Omega_\mathbf{N}\}$ as follows: $\chi_1 \approx \chi_2$ iff $\chi_1(\theta) = \chi_2(\theta)$, for all $\theta \in \Theta_\mathcal{N}$. Let $\Upsilon_\approx$ denote the set of $\approx$-equivalent classes of Υ, and $\gamma \in \Upsilon_\approx$. Clearly, for each $\chi \in \gamma$ we have the same $\mathcal{N}^\chi$ (resp. $\lambda^\chi_{S,v,v'}$), so we can denote it as $\mathcal{N}^\gamma$ (resp. $\lambda^\gamma_{S,v,v'}$). To symbolically represent parameter valuations on Θ_φ, we construct for each $\gamma \in \Upsilon_\approx$ an RA-formula $\Delta(\varphi, v^*, \gamma)$, with free variables $\theta_1, \ldots, \theta_k \in \Theta_\varphi$, such that $\mathcal{N}^\chi, v^* \models \varphi^\chi$ for some valuation $\chi \in \gamma$ iff the sentence $\exists\theta_1 \ldots \exists\theta_k \Delta(\varphi, v^*, \chi)$ is true. The approach is correct because RA has a decidable theory and $\Upsilon_\approx$ is a finite set. The main instrument of the approach is to describe by an RA-formula, for given two regions $v = [q]$, $v' = [q']$ in $G(\mathcal{N}^\gamma)$, all the possible values of duration $D(r, q_j)$ for finite runs r from q to q' in $\mathcal{N}^\gamma$. For a region v of G^γ and a PTCTL-formula φ, the construction of $\Delta(\varphi, v, \gamma)$ is easily performed by induction on the length of φ.

Theorem 1. *Given $\gamma \in \Upsilon_\approx$, a region v of $\mathcal{N}^\gamma$, a PTCTL-formula φ with $\Theta_\varphi = \{\theta_1, ..., \theta_k\}$, there exists an RA-formula $\Delta(\varphi, v, \gamma)$ such that $\mathcal{N}^\chi, v \models \varphi^\chi$ for some valuation $\chi \in \gamma$ iff the sentence $\exists\theta_1 ... \exists\theta_k \Delta(\varphi, v, \gamma)$ is true. The construction of $\Delta(\varphi, v, \gamma)$ is effective.*

Theorem 2. *There exists a procedure for solving $\mathcal{PTBA}(\mathcal{N}_\Omega, \varphi)$ which is in 2ExpTime in the product of the sizes of $\mathcal{N}$ and φ.*

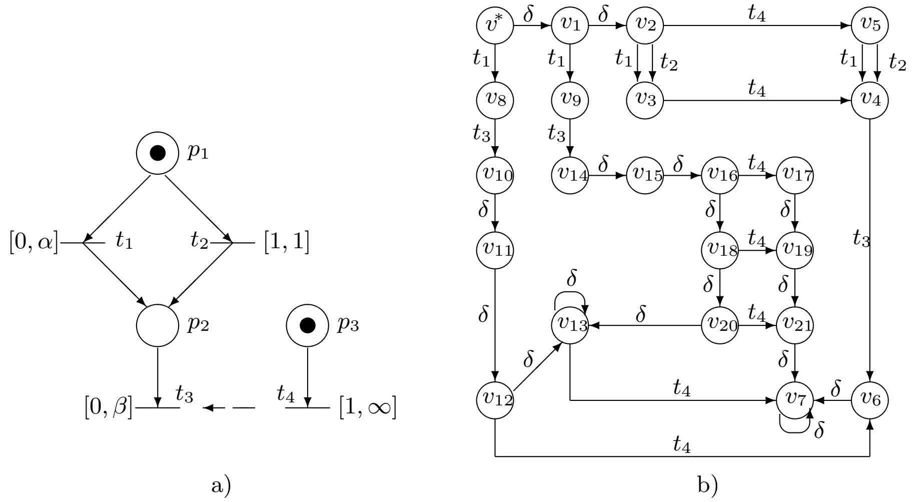


Fig. 1. a) PPrTPN $\mathcal{N}_1$, b) region graph $G(\mathcal{N}_1^\gamma)$

Example 1. In Fig. 1a), an example of a PPrTPN $\mathcal{N}_1$ is given that includes two linear terms $\alpha = \theta_1 + 1$ and $\beta = \theta_2$. Assume $\Omega : \theta_1 = 0,\ 0 \leq \theta_2 \leq 1$. Consider the case with $\mathcal{N}_1^\gamma$, where $\gamma = \{\chi \mid \chi(\theta_1) = 0,\ \chi(\theta_2) = 0\} \in \Upsilon_\approx$. The region graph $G(\mathcal{N}_1^\gamma)$ is shown in Fig. 1b). Contemplate the PTCTL-formula $\varphi = \forall\ \Box_{>\theta}\ (m(p_2) = 0\ \vee\ m(p_3) = 0)$. Applying standard transformations, we get $\varphi = \neg(true \exists\ U_{>\theta}(m(p_2) > 0 \wedge m(p_3) > 0))$. Using the reasonings in the proof of Theorem 1 [10], $\Delta(true \exists U_{>\theta}(m(p_2) > 0 \wedge m(p_3) > 0), v^*, \gamma) = [\Delta(m(p_2) > 0 \wedge m(p_3) > 0, v^*, \gamma) \wedge (0 > \theta)] \bigvee_{v' \in V} \bigvee_{S \subset V} [\exists x > \theta \lambda^\gamma_{S,v^*,v'}(x) \wedge \Delta(m(p_2) > 0 \wedge m(p_3) > 0, v', \gamma) \wedge \bigwedge_{s \in S} \Delta(true, s, \gamma) \wedge (\neg B(v') \rightarrow \Delta(true, v', \gamma))]$. One can see that $\Delta(m(p_2) > 0 \wedge m(p_3) > 0, v', \gamma)$ is true only for $v' = v_3$, $v' = v_8$ and $v' = v_9$. Then, $\lambda^\gamma_{S,v^*,v_3}(x) =$ "$x = 1$", $\lambda^\gamma_{S,v^*,v_8}(x) =$ "$x = 0$", $\lambda^\gamma_{S,v^*,v_9}(x) =$ "$0 < x < 1$", for all $S \subseteq V$ such that $\lambda^\gamma_{S,v^*,v'} \neq \emptyset$. Thus, we have $\Delta(true \exists U_{>\theta}(m(p_2) > 0 \wedge m(p_3) > 0), v^*, \gamma) = \exists x > \theta\ 0 \leq x \leq 1$. So, $\Delta(\forall \Box_{>\theta}(m(p_2) = 0\ \vee\ m(p_3) = 0), v^*, \gamma) = \neg(\exists x > \theta\ \ 0 \leq x \leq 1)$, i.e. $\theta \geq 1$. For the other possible $\gamma \in \Upsilon_\approx$, the results are obtained analogously.

References

1. Alur, R., Dill, D.: The theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Alur, R., Etessami, K., La Torre, S., Peled, D.: Parametric temporal logic for "model measuring". In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 159–168. Springer, Heidelberg (1999)
3. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Proc. STOC 1993, pp. 592–601. ACM Press, New York (1993)
4. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of the expressiveness of timed automata and time Petri nets. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 211–225. Springer, Heidelberg (2005)
5. Berthomieu, B., Peres, F., Vernadat, F.: Bridging the gap between timed automata and bounded time petri nets. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 82–97. Springer, Heidelberg (2006)
6. Bruyère, V., Dall'olio, E., Raskin, J.-F.: Durations, parametric model-checking in timed automata with Pressburger arithmetic. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 687–698. Springer, Heidelberg (2003)
7. Bruyère, V., Raskin, J.-F.: Real-time model-checking: Parameters everywhere. Logical Methods in Computer Science 3(1:7), 1–30 (2007)
8. Merlin, P., Faber, D.J.: Recoverability of communication protocols. IEEE Trans. of Communication COM-24(9) (1976)
9. Louis-Marie Traonouez, L.-M., Lime, D., Roux, O.H.: Parametric model-checking of time petri nets with stopwatches using the state-class graph. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 280–294. Springer, Heidelberg (2008)
10. Virbitskaite, I.B., Dedova, A.V.: Towards Parametric Verification of Prioritized Time Petri Nets, `http://www.iis.nsk.su/persons/virb/virbded09.zip`
11. Virbitskaite, I.B., Pokozy, E.A.: Parametric behaviour analysis for time Petri nets. In: Malyshkin, V.E. (ed.) PaCT 1999. LNCS, vol. 1662, pp. 134–140. Springer, Heidelberg (1999)
12. Wang, F.: Parametric timing analysis for real-time systems. Information and Computation 130, 131–150 (1996)

Software Transactional Memories: An Approach for Multicore Programming

Damien Imbs and Michel Raynal

IRISA, Université de Rennes 1, 35042 Rennes, France
{damien.imbs,raynal}@irisa.fr

Abstract. The recent advance of multicore architectures and the deployment of multiprocessors as the mainstream computing platforms have given rise to a new concurrent programming impetus. Software transactional memories (STM) are one of the most promising approach to take up this challenge. The aim of a STM system is to discharge the application programmer from the management of synchronization when he/she has to write multiprocess programs. His/her task is to decompose his/her program in a set of sequential tasks that access shared objects, and to decompose each task in atomic units of computation. The management of the required synchronization is ensured by the associated STM system. This paper presents two STM systems, and a formal proof for the second one. Such a proof -that is not trivial- is one of the very first proofs of a STM system. In that sense, this paper strives to contribute to the establishment of theoretical foundations for STM systems.

Keywords: Concurrent programming, Consistent global state, Consistency condition, Linearizability, Lock, Logical clock, Opacity, Serializability, Shared object, Software transactional memory, Transaction.

1 Introduction

The challenging advent of multicore architectures. The speed of light has a limit. When combined with other physical and architectural demands, this physical constraint places limits on processor clocks: their speed is no longer rising. Hence, software performance can no longer be obtained by increasing CPU clock frequencies. To face this new challenge, (since a few years ago) manufacturers have investigated and are producing what they call *multicore architectures*, i.e., architectures in which each chip is made up of several processors that share a common memory. This constitutes what is called "the multicore revolution" [6].

The main challenge associated with multicore architectures is "how to exploit their power?" Of course, the old (classical) "multi-process programming" (multi-threading) methods are an answer to this question. Basically, these methods provide the programmers with the concept of a *lock*. According to the abstraction level considered, this lock can be a semaphore object, a monitor object, or the programmer's favorite synchronization object.

Unfortunately, traditional lock-based solutions have inherent drawbacks. On one side, if the set of data whose accesses are controlled by a single lock is too large

(large grain), the parallelism can be drastically reduced. On another side, the solutions where a lock is associated with each datum (fine grain), are error-prone (possible presence of subtle deadlocks), difficult to design, master and prove correct. In other words, providing the application programmers with locks is far from being the panacea when one has to produce correct and efficient multi-process (multi-thread) programs. Interestingly enough, multicore architectures have (in some sense) rang the revival of concurrent programming.

The Software Transactional Memory approach. The concept of *Software Transactional Memory* (STM) is an answer to the previous challenge. The notion of transactional memory has first been proposed (fifteen years ago) by Herlihy and Moss to implement concurrent data structures [7]. It has then been implemented in software by Shavit and Touitou [15], and has recently gained a great momentum as a promising alternative to locks in concurrent programming [3,5,12,14].

Transactional memory abstracts the complexity associated with concurrent accesses to shared data by replacing locking with atomic execution units. In that way, the programmer has to focus where atomicity is required and not on the way it has to be realized. The aim of a STM system is consequently to discharge the programmer from the direct management of synchronization entailed by accesses to concurrent objects.

More generally, STM is a middleware approach that provides the programmers with the *transaction* concept (this concept is close but different from the notion of transactions encountered in databases [3]). More precisely, a process is designed as (or decomposed into) a sequence of transactions, each transaction being a piece of code that, while accessing any number of shared objects, always appears as being executed atomically. The job of the programmer is only to define the units of computation that are the transactions. He does not have to worry about the fact that the base objects can be concurrently accessed by transactions. Except when he defines the beginning and the end of a transaction, the programmer is not concerned by synchronization. It is the job of the STM system to ensure that transactions execute as if they were atomic.

Of course, a solution in which a single transaction executes at a time trivially implements transaction atomicity but is irrelevant from an efficiency point of view. So, a STM system has to do "its best" to execute as many transactions per time unit as possible. Similarly to a scheduler, a STM system is an on-line algorithm that does not know the future. If the STM is not trivial (i.e., it allows several transactions that access the same objects in a conflicting manner to run concurrently), this intrinsic limitation can direct it to abort some transactions in order to ensure both transaction atomicity and object consistency. From a programming point of view, an aborted transaction has no effect (it is up to the process that issued an aborted transaction to re-issue it or not; usually, a transaction that is restarted is considered as a new transaction).

Content of the paper. This paper is an introduction to STM systems, with an algorithmic and theoretical flavor. It is made up of three main sections. First, Section 2 presents a consistency condition suited to STM systems (called opacity [4]), and a STM architecture. Then, each of the two following sections presents a STM system that -in its own way- ensures the opacity consistency condition. The first (Section 3) is a simplified version of the well-known TL2 system [2] that has proved to be particularly efficient on meaningful benchmarks. The design of nearly all the STM systems proposed so far has been driven by efficiency, and nearly none of them has been proved correct. So, to give a broader view of the STM topic, the second STM system that is presented (Section 4) is formally proved correct[1]. Formal proofs are important as they provide STM systems with solid foundations, and consequently participate in establishing STM systems as a fundamental approach for concurrent programming [1].

2 A STM Computation Model

2.1 On STM Consistency Conditions

The classical consistency criterion for database transactions is serializability [13] (sometimes strengthened in "strict serializability", as implemented when using the 2-phase locking mechanism). The serializability consistency criterion involves only the transactions that are committed. Said differently, a transaction that aborts is not prevented from accessing an inconsistent state before aborting. In a STM system, the code encapsulated in a transaction can be any piece of code (involving shared data), it is not restricted to predefined patterns. Consequently a transaction always has to operate on a consistent state. To be more explicit, let us consider the following example where a transaction contains the statement $x \leftarrow a/(b-c)$ (where a, b and c are integer data), and let us assume that $b - c$ is different from 0 in all the consistent states. If the values of b and c read by a transaction come from different states, it is possible that the transaction obtains values such as $b = c$ (and $b = c$ defines an inconsistent state). If this occurs, the transaction raises an exception that has to be handled by the process that invoked the corresponding transaction[2]. Such bad behaviors have to be prevented in STM systems: whatever its fate (commit or abort) a transaction has to always see a consistent state of the data it accesses. The aborted transactions have to be harmless.

This is what is captured by the *opacity* consistency condition. Informally suggested in [2], and then formally introduced and deeply investigated in [4], opacity requires that no transaction reads values from an inconsistent global state. It is strict serializability when considering each committed transaction entirely, and an appropriate read prefix of each aborted transaction.

More precisely, let us associate with each aborted transaction T the read prefix that contains all its read operations until T aborts (if the abort is entailed by a read, this read is not included in the prefix). An execution of a set of transactions satisfies the *opacity* condition if all the committed transactions and the read prefix of each aborted transaction appear as if they have been executed one after the other, this sequential order being in agreement with their real time occurrence order. A formal definition of opacity appears in [4]. A very general framework for consistency conditions is introduced in [10], where is also introduced the *virtual world consistency* condition. A protocol implementing this general condition is described in [11].

[1] The STM system presented in this section has been introduced in [9] without being proved correct. So, this section validates and complements that paper.

[2] Even worse undesirable behaviors can be obtained when reading values from inconsistent states. This occurs for example when an inconsistent state provides a transaction with values that generate infinite loops.

2.2 The STM System Interface

The STM system provides the transactions with four operations denoted $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$, where T is a transaction, and X a shared base object.

- $\mathsf{begin}_T()$ is invoked by T when it starts. It initializes local control variables.
- $X.\mathsf{read}_T()$ is invoked by the transaction T to read the base object X. That operation returns a value of X or the control value $abort$. If $abort$ is returned, the invoking transaction is aborted (in that case, the corresponding read does not belong to the read prefix associated with T).
- $X.\mathsf{write}_T(v)$ is invoked by the transaction T to update X to the new value v. That operation returns the control value ok or the control value $abort$. Like in the operation $X.\mathsf{read}_T()$, if $abort$ is returned, the invoking transaction is aborted.
- If a transaction attains its last statement (as defined by the user, which means it has not been aborted before) it executes the operation $\mathsf{try\_to\_commit}_T()$. That operation decides the fate of T by returning $commit$ or $abort$. (Let us notice, a transaction T that invokes $\mathsf{try\_to\_commit}_T()$ has not been aborted during an invocation of $X.\mathsf{read}_T()$.)

2.3 The Incremental Read/Deferred Update Model

In this transaction system model, each transaction T uses a local working space. When T invokes $X.\mathsf{read}_T()$ for the first time, it reads the value of X from the shared memory and copies it into its local working space. Later $X.\mathsf{read}_T()$ invocations (if any) use this copy. So, if T reads X and then Y, these reads are done incrementally, and the state of the shared memory can have changed in between.

When T invokes $X.\mathsf{write}_T(v)$, it writes v into its working space (and does not access the shared memory). Finally, if T is not aborted while it is executing $\mathsf{try\_to\_commit}_T()$, it copies the values written (if any) from its local working space to the shared memory. (A similar deferred update model is used in some database transaction systems.)

3 A Sketch of TL2 (Transactional Locking 2)

3.1 Aim and Principles

The TL2 STM system [2] aims at reducing the synchronization cost due to the read, write and validation (i.e., $\mathsf{try\_to\_commit}()$) operations. To that end, it associates a lock with each data object and uses a logical global clock (integer) that is read by all the transactions and increased by each writing transaction that commits. This global clock is basically used to validate the consistency of the state of the data objects a transaction is working on. The TL2 protocol is particularly efficient when there are few conflicts between concurrent transactions. (Two transactions conflict if they concurrently access the same object and one access is a write).

TL2 ensures the opacity property. The performance study depicted in [2] (based on a red-black tree benchmark) shows that TL2 is pretty efficient. It has nevertheless scenarios in which a transaction is directed to abort despite the fact that it has read values from a consistent state (these scenarios depend on the value of the global clock. They can occur when, despite the fact that all the values read by a transaction T are mutually consistent, one of them has been written by a transaction concurrent with T).

3.2 A Simplified Version of TL2

A very schematic description of the operations $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$, $X.\mathsf{write}_T(v)$ and $\mathsf{try\_to\_commit}_T()$ used in TL2 [2] is given in Figure 1 for an update transaction. As indicated previously, the protocols implementing these operations are based on a global (logical) clock and a lock per object.

The global clock, denoted $CLOCK$, is increased (atomic $Fetch\&Increment()$ operation) each time an update transaction T invokes $\mathsf{try\_to\_commit}_T()$ (line 11). Moreover, when a transaction starts it invokes the additional operation $\mathsf{begin}_T()$ to obtain a birthdate (defined as the current value of the clock).

At the implementation level, an object X is made up of two fields: a data field $X.value$ containing the current value of the object, and a control field $X.date$ containing the date at which that value was created (line 12 of $\mathsf{try\_to\_commit}_T()$). A lock is associated with each object.

The case of an update transaction. Each transaction T manages a local read set lrs_T, and a local write set lws_T. As far as a $X.\mathsf{read}_T()$ is concerned we have the following. If, previously, a local copy lcx of the object X has been created by an invocation of $X.\mathsf{write}_T(v)$ issued by the same transaction, its value is returned (lines 01-02). Otherwise, X is read from the shared memory, and X's id is added to the local read set lrs_T (line 03). Finally, if the date associated with the current value of X is greater than the birthdate of T, the transaction is aborted (line 04). (This is because, T has possibly read other values that are no longer consistent with the value of X just obtained.) If the date associated with the current value of X is not greater than the birthdate of T, that value is returned by the $X.\mathsf{read}_T()$ operation. (In that case, the value read is consistent with the values previously read by T.)

The operation $X.\mathsf{write}_T(v)$ in TL2 and the one in the proposed protocol are similar. If there is no local copy of X, one is created and its value field is set to v. The local write set lws_T is also updated to remember that X has been written by T. The lifetime of the local copy lcx of X created by a $X.\mathsf{write}_T(v)$ operation spans the duration of the transaction T.

When a transaction T invokes $\mathsf{try\_to\_commit}_T()$ it first locks all the objects in $lrs_T \cup lws_T$. Then, T checks if the current values of the objects X it has read are still mutually consistent, and consistent with respect to the new values it has (locally) written. This is done by comparing the current date $X.date$ of each object X that has been read to the birthdate of T. If one of these dates is greater than its birthdate, there is a possible inconsistency and consequently T is aborted (line 11). Otherwise, T can be committed. Before being committed (line 14), T has to set the objects it has written to their new values (line 13). Their control part has also to be updated: they are set to the last clock value obtained by T (line 12). Finally, T releases the locks and commits.

Remark. This presentation of the $\mathsf{try\_to\_commit}_T()$ operation of TL2 does not take into account all of its aspects. As an example, if at line 09, all the locks cannot be

```
operation begin_T(): $birthdate \leftarrow CLOCK$.
==================================================================
operation $X$.read_T():
(01) if (there is a local copy $lcx$ of $X$)
(02)    then return $(lcx.value)$ % the local copy $lcx$ was created by a write of $X$ %
(03)    else $lcx \leftarrow$ copy of $X$ read from the shared memory; $lrs_T \leftarrow lrs_T \cup \{X\}$;
(04)         if $lcx.date > birthdate$ then return $(abort)$ else return $(lcx.value)$ end if
(05) end if.
==================================================================
operation $X$.write_T($v$):
(06) if (there is no local copy of $X$) then allocate local space $lcx$ for a copy end if;
(07) $lcx.value \leftarrow v$; $lws_T \leftarrow lws_T \cup \{X\}$;
(08) return $(ok)$.
==================================================================
operation try_to_commit_T():
(09)    lock all the objects in $(lrs_T \cup lws_T)$;
(10)    for each $X \in lrs_T$ do % the date of $X$ is read from the shared memory %
(11)        if $X.date > birthdate$ then release all the locks; return $(abort)$ end if end for;
(12)    $commit\_date \leftarrow Fetch\&Increment(CLOCK)$;
(13)    for each $X \in lws_T$ do $X \leftarrow (lcx.value, commit\_date)$ end for;
(14)    release all the locks; return $(commit)$.
```

Fig. 1. TL2 algorithm for an update transaction

immediately obtained, TL2 can abort the transaction (and restart it later). This can allow for more efficient behaviors. Moreover, the lock of an object is used to contain its date value (this allows for more efficient read operations.)

The Case of a read only transaction. Such a transaction T does not modify the shared objects. The code of its X.read$_T$() and try_to_commit$_T$() operations can be simplified. This is left as an exercise for the reader.

4 A Window-Based STM System

This STM system has been proposed in [9] where is introduced the notion of obligation property. It is called *window-based* because a logical time window is associated with each transaction, and a transaction has to be aborted when its window becomes empty. The opacity property does not prevent a STM system from aborting all the transactions. An obligation property states circumstances in which a STM system must commit a transaction T. Two obligation properties are defined in [9], and the aim of the system described in [9] is to satisfy both opacity and these obligation properties.

4.1 The STM Control Variables

The object fields, the object locks, the logical global clock, the local sets lrs_T and lws_T have the same meaning as in TL2. The following additional (shared or local) control variables are also used:

- A set RS_X per base object X. This set, initialized to $\emptyset$, contains the ids of the transactions that have read X since the last update of X. A transaction adds its id to RS_X to indicate a possible read/write conflict.
- A control variable MAX_DATE_T, initialized to $+\infty$, is associated with each transaction T. It keeps the smallest date at which an object read by T has been overwritten. That variable allows the transaction T to safely evaluate the abstract property $P2(T)$. As we will see, we have $P2(T) \Rightarrow (MAX\_DATE_T = +\infty)$, and the STM system will direct T to commit when $MAX\_DATE_T = +\infty$.
- $read\_only_T$ is a local boolean, initialized to $true$, that is set to $false$, if T invokes a $X.\mathsf{write}_T(v)$ operation.
- min_date_T is a local variable containing the greatest date of the objects T has read so far. Its initial value is 0. Combined with MAX_DATE_T, that variable allows a safe evaluation of the abstract property $P1(T)$. As we will see, we have $P1(T) \Rightarrow (min\_date_T \leq MAX\_DATE_T)$, and the STM system will not abort a read-only transaction T if $min\_date_T \leq MAX\_DATE_T$.

4.2 The STM Operations

The three operations that constitute the STM system $X.\mathsf{read}_T()$, $X.\mathsf{write}_T(v)$, and $\mathsf{try\_to\_commit}_T()$, are described in Figure 2. As in a lot of other protocols (e.g., STM or discrete event simulation), the underlying idea is to associate a time window, namely $[min\_date_T, MAX\_DATE_T]$, with each transaction T. This time window is managed as follows:

- When a read-only or update transaction T reads a new object (from the shared memory), it accordingly updates min_date_T, and aborts if its time window becomes empty. A time window becomes empty when the system is unable to guarantee that the values previously read by T and the value it has just obtained belong to a consistent snapshot.
- When an update transaction T is about to commit, it has two things to do. First, write into the shared memory the new values of the objects it has updated, and define their dates as the current clock value. These writes may render inconsistent the snapshot of a transaction T' that has already obtained values and will read a new object in the future. Hence, in order to prevent such an inconsistency from occurring (see the previous item), the transaction T sets $MAX\_DATE_{T'}$ to the current clock value if $\big((T' \in RS_X) \wedge (X \in lws_T)\big)$ and $(MAX\_DATE_{T'} = +\infty)$.

The operation $X.\mathsf{read}_T()$. When T invokes $X.\mathsf{read}_T()$, it obtains the value of X currently kept in the local memory if there is one (lines 01 and 07). Otherwise, T first allocates space in its local memory for a copy of X (line 02), obtains the value of X from the shared memory and updates RS_X accordingly (line 03). The update of RS_X allows T to announce a read/write conflict that will occur with the transactions that will update X. This line is the only place where read/write conflicts are announced in the proposed STM algorithm.

Then, T updates its local control variables lrs_T and min_date_T (line 04) in order to keep them consistent. Finally, T checks its time window (line 05) to know if its snapshot is consistent. If the time window is empty, the value it has just obtained from the memory can make its current snapshot inconsistent and consequently T aborts.

Remark. Looking into the details, when a transaction T reads X from the shared memory, a single cause can cause the window predicate $(min\_date_T > MAX\_DATE_T)$ to be true: min_date_T has just been increased, and MAX_DATE_T has been decreased to a finite value. T is then aborted due to a write/read conflict on X and a read/write conflict on $Y \neq X$.

```
operation $X$.read$_T$():
(01) if (there is no local copy of $X$) then
(02)    allocate local space $lcx$ for a copy;
(03)    lock $X$; $lcx \leftarrow X$; $RS_X \leftarrow RS_X \cup \{T\}$; unlock $X$;
(04)    $lrs_T \leftarrow lrs_T \cup \{X\}$; $min\_date_T \leftarrow \max(min\_date_T, lcx.date)$;
(05)    if $(min\_date_T > MAX\_DATE_T)$ then return$(abort)$ end if
(06) end if;
(07) return $(lcx.value)$.
==================================================================
operation $X$.write$_T(v)$:
(08) $read\_only_T \leftarrow false$;
(09) if (there is no local copy of $X$) then allocate local space $lcx$ for a copy end if;
(10) $lcx.value \leftarrow v$; $lws_T \leftarrow lws_T \cup \{X\}$;
(11) return $(ok)$.
==================================================================
operation try_to_commit$_T$():
(12) if $(read\_only_T)$
(13)    then return$(commit)$
(14)    else lock all the objects in $lrs_T \cup lws_T$;
(15)         if $(MAX\_DATE_T \neq +\infty)$ then release all the locks; return$(abort)$ end if;
(16)         $current\_time \leftarrow CLOCK$;
(17)         for each $T' \in \big(\cup_{X \in lws_T} RS_X\big)$
                     do $C\&S(MAX\_DATE_{T'}, +\infty, current\_time)$ end for;
(18)         $commit\_time \leftarrow Fetch\&Increment(CLOCK)$;
(19)         for each $X \in lws_T$ do $X \leftarrow (lcx.value, commit\_time)$; $RS_X \leftarrow \emptyset$ end for;
(20)         release all the locks; return$(commit)$
(21) end if.
```

Fig. 2. A window-based STM system

The operation X.write$_T()$. The text of the algorithm implementing X.write$_T()$ is very simple. The transaction first sets a flag to record that it is not a read-only transaction (line 08). If there is no local copy of X, corresponding space is allocated in the local memory (line 09); let us remark that this does not entail a read of X from the shared memory. Finally, T updates the local copy of X, and records in lrw_T that it has locally written the copy of X (line 10). It is important to notice that an invocation of X.write$_T()$ is purely local: it involves no access to the shared memory, and cannot entail an immediate abort of the corresponding transaction.

The operation $\text{try\_to\_commit}_T()$. This operation works as follows. If the invoking transaction is a read-only transaction, it is committed (lines 12-13). So, a read-only transaction can abort only during the invocation of a $X.\text{read}_T()$ operation (line 05).

If the transaction T is an update transaction, $\text{try\_to\_commit}_T()$ first locks all the objects accessed by T (line 14). (In order to prevent deadlocks, it is assumed that these objects are locked according to a predefined total order, e.g., their identity order.) Then, T checks if $MAX\_DATE_T \neq +\infty$. If this is the case, there is a read/write conflict: T has read an object that since then has been overwritten. Consequently, there is no guarantee for the current snapshot of T (that is consistent) and the write operations of T to appear as being atomic. T consequently aborts (after having released all the locks it has previously acquired, line 15).

If the predicate $MAX\_DATE_T = +\infty$ is true, T will necessarily commit. But, before releasing the locks and committing (line 20), T has to (1) write in the shared memory the new values of the objects with their new dates (lines 18-19), and (2) update the control variables to indicate possible (read/write with read in the past, or write/read with read in the future) conflicts due to the objects it has written. As indicated at the beginning of this section, (1) read/write conflicts are managed by setting $MAX\_DATE_{T'}$ to the current clock value for all the transactions T' such that $\big((T' \in RS_X) \wedge (X \in lws_T)\big)$ (lines 16-17), and consequently RS_X is reset to $\emptyset$ (line 19), while (2) write/read conflicts on an object X are managed by setting the date of X to the commit time of T.

As two transactions $T1$ and $T2$ can simultaneously find $MAX\_DATE_{T'} = +\infty$ and try to change its value, the modification of $MAX\_DATE_{T'}$ is controlled by an atomic compare&swap operation (denoted $C\&S()$, line 17).

4.3 Formal Framework to Prove the Opacity Property

Events at the shared memory level. Each transaction generates events defined as follows.

- Begin and end events. The event denoted B_T is associated with the beginning of the transaction T, while the event E_T is associated with its termination. E_T can be of two types, namely A_T and C_T, where A_T is the event "abort of T", while C_T is the event "commit of T".
- Read events. The event denoted $r_T(X)v$ is associated with the atomic read of X (from the shared memory) issued by the transaction T. The value v denotes the value returned by the read. If the value v is irrelevant $r_T(X)v$ is abbreviated $r_T(X)$.
- Write events. The event denoted $w_T(X)v$ is associated with the atomic write of the value v in the shared object X (in the shared memory). If the value v is irrelevant $w_T(X)v$ is abbreviated $w_T(X)$. Without loss of generality we assume that no two writes on the same object X write the same value. We also assume that all the objects are initially written by a fictitious transaction.

History at the shared memory level. Given an execution, let H be the set of all the (begin, end, read and write) events generated by the transactions. As the events correspond to atomic operations, they can be totally ordered. It follows that, at the shared memory level, an execution can be represented by the pair $\widehat{H} = (H, <_H)$ where $<_H$

denotes the total ordering on its events. $\widehat{H}$ is called a *shared memory history*. As $<_H$ is a total order, it is possible to associate a unique "date" with each event in H. (In the following an event is sometimes used to denote its date.)

History at the transaction level. Let TR be the set of transactions issued during an execution. Let $\rightarrow_{TR}$ be the order relation defined on the transactions of TR as follows: $T1 \rightarrow_{TR} T2$ if $E_{T1} <_H B_{T2}$ ($T1$ has terminated before $T2$ starts). If $T1 \not\rightarrow_{TR} T2 \wedge T2 \not\rightarrow_{TR} T1$, we say that $T1$ and $T2$ are concurrent (their executions overlap in time). At the transaction level, that execution is defined by the partial order $\widehat{TR} = (TR, \rightarrow_{TR})$, that is called a *transaction level history* or a *transaction run*.

Sequential, equivalent and linearizable histories. A transaction history $\widehat{ST} = (ST, \rightarrow_{ST})$ is *sequential* if no two of its transactions are concurrent. Hence, in a sequential history, $T1 \not\rightarrow_{ST} T2 \Leftrightarrow T2 \rightarrow_{ST} T1$, thus $\rightarrow_{ST}$ is a total order. A sequential transaction history is *legal* if each of its read operations returns the value of the last write on the same object.

A sequential transaction history $\widehat{ST}$ is *equivalent* to a transaction history $\widehat{TR}$ if (1) $ST = TR$ (i.e., they are made of the same transactions (same values read and written) in $\widehat{ST}$ and in $\widehat{TR}$), and (2) the total order $\rightarrow_{ST}$ respects the partial order $\rightarrow_{TR}$ (i.e., $\rightarrow_{TR} \subseteq \rightarrow_{ST}$).

A transaction history $\widehat{AA}$ is *linearizable* if there exists a history $\widehat{SA}$ that is sequential, legal and equivalent to $\widehat{AA}$ [8]. If a transaction history $\widehat{AA}$ is linearizable it is possible to associate a single point of the time line with every transaction, no two transactions being associated with the same point. This point is called the *linearization point* of the corresponding transaction.

Reduced histories. Given a run of transactions $\widehat{TR} = \big(TR, \rightarrow_{TR}\big)$, let $\mathcal{C}$ (resp. $\mathcal{A}$) be the set of transactions that commit (resp., abort) in that run.

Given $T \in \mathcal{A}$, let $T' = \rho(T)$ be the transaction built from T as follows (ρ stands for "reduced"). As T has been aborted, there is a read or a write on a base object that entailed that abortion. Let $\mathit{prefix}(T)$ be the prefix of T that includes all the read and write operations on the base objects accessed by T until (but excluding) the read or write that entailed the abort of T. $T' = \rho(T)$ is obtained from $\mathit{prefix}(T)$ by replacing its write operations on base objects and all the subsequent read operations on these objects, by corresponding write and read operations on a copy in local memory. The idea here is that only an appropriate prefix of an aborted transaction is considered: its write operations on base objects (and the subsequent read operations) are made fictitious in $T' = \rho(T)$.

Finally, let $\mathcal{A}' = \{T' \mid T' = \rho(T) \ \wedge \ T \in \mathcal{A}\}$, and $\widehat{\rho(TR)} = \big(\rho(TR), \rightarrow_{\rho(TR)}\big)$ where $\rho(TR) = \mathcal{C} \cup \mathcal{A}'$ (i.e., $\rho(TR)$ contains all the transactions of $\widehat{TR}$ that commit, plus $\rho(T)$ for each transaction $T \in TR$ that aborts) and $\rightarrow_{\rho(TR)} = \rightarrow_{TR}$. Opacity states that the transactions in $\mathcal{C} \cup \mathcal{A}'$ can be consistently and totally ordered according to their real-time order, i.e., $\widehat{\rho(TR)}$ is linearizable.

Types of conflict. Two operations conflict if both access the same object and one of these operations is a write. Considering two transactions $T1$ and $T2$ that access the same object X, three types of conflict can occur. More specifically:

- Read/write conflict: $\mathit{conflict}(X, R_{T1}, W_{T2}) \overset{\text{def}}{=} \big(r_{T1}(X) <_H w_{T2}(X)\big)$.
- Write/read conflict: $\mathit{conflict}(X, W_{T1}, R_{T2}) \overset{\text{def}}{=} \big(w_{T1}(X) <_H r_{T2}(X)\big)$.
- Write/write conflict: $\mathit{conflict}(X, W_{T1}, W_{T2}) \overset{\text{def}}{=} \big(w_{T1}(X) <_H w_{T2}(X)\big)$.

The read-from relation. The *read-from* relation between transactions, denoted $\rightarrow_{rf}$, is defined as follows: $T1 \overset{X}{\rightarrow}_{rf} T2$ if $T2$ reads the value that $T1$ wrote in the object X.

4.4 A Formal Proof of the Opacity Property

Principle of the proof of the opacity property. According to the algorithms implementing the operations $X.\mathsf{read}_T()$ and $X.\mathsf{write}_T(v)$ described in Figure 2, we ignore all the read operations on an object that follow another operation on the same object within the same transaction, and all the write operations that follow another write operation on the same object within the same transaction (these are operations local to the memory of the process that executes them). Building $\rho(\mathit{TR})$ from TR is then a straightforward process.

To prove that the protocol described in Figure 2 satisfies the opacity consistency criterion, we need to prove that, for any transaction history $\widehat{\mathit{TR}}$ produced by this protocol, there is a sequential legal history $\widehat{\mathit{ST}}$ equivalent to $\widehat{\rho(\mathit{TR})}$. This amounts to prove the following properties (where $\widehat{H}$ is the shared memory level history generated by the transaction history $\widehat{\mathit{TR}}$):

1. $\rightarrow_{ST}$ is a total order,
2. $\forall T \in \mathit{TR} : \big(T \text{ commits} \Rightarrow T \in \mathit{ST}\big) \wedge \big(T \text{ aborts} \Rightarrow \rho(T) \in \mathit{ST}\big)$,
3. $\rightarrow_{\rho(\mathit{TR})} \subseteq \rightarrow_{ST}$,
4. $\mathit{T1} \overset{X}{\rightarrow}_{rf} \mathit{T2} \Rightarrow \nexists \mathit{T3}$ such that $\big(\mathit{T1} \rightarrow_{ST} \mathit{T3} \rightarrow_{ST} \mathit{T2}\big) \wedge \big(w_{\mathit{T3}}(X) \in H\big)$,
5. $\mathit{T1} \overset{X}{\rightarrow}_{rf} \mathit{T2} \Rightarrow \mathit{T1} \rightarrow_{ST} \mathit{T2}$.

Definition of the linearization points. ST is produced by ordering the transactions according to their linearization points. The linearization point of the transaction T is denoted ℓ_T. The linearization points of the transactions are defined as follows :

- If a transaction T aborts, ℓ_T is the time at which its $\mathit{MAX\_DATE}_T$ global variable is assigned a finite value by a transaction T' (line 17 of the $\mathsf{try\_to\_commit}()$ operation of T').
- If a read-only transaction T commits, ℓ_T is placed at the earliest of (1) the occurrence time of the test during its last read operation (line 05 of the $X.\mathsf{read}()$ operation) and (2) the time at which $\mathit{MAX\_DATE}_T$ is assigned a finite value by another transaction. This value is unique and well-defined (this follows from the invocation of $C\&S(\mathit{MAX\_DATE}_{T'}, +\infty, \mathit{current\_time})$ at line 17).
- If an update transaction T commits, ℓ_T is placed at the execution of line 18 by T (read and increase of the clock).

The total order $<_H$ (defined on the events generated by $\widehat{\mathit{TR}}$) can be extended with these linearization points. Transactions whose linearization points happen at the same time are ordered arbitrarily.

Proof of the opacity property. Let $\widehat{TR} = (TR, \rightarrow_{TR})$ be a transaction history. Let $\widehat{ST} = (\rho(TR), \rightarrow_{ST})$ be a history whose transactions are the transactions $\rho(TR)$, and such that $\rightarrow_{ST}$ is defined according to the linearization points of each transaction in $\rho(TR)$. If two transactions have the same linearization point, they are ordered arbitrarily. Finally, let us recall that the linearization points can be trivially added to the sequential history $\widehat{H} = (H, <_H)$ defined on the events generated by the transaction history $\widehat{TR}$. So, we consider in the following that the set H includes the transaction linearization points.

Lemma 1. *$\rightarrow_{ST}$ is a total order.*

Proof. Trivial from the definition of the linearization points. □

Lemma 2. $\rightarrow_{\rho(TR)} \subseteq \rightarrow_{ST}$.

Proof. This lemma follows from the fact that, given any transaction T, its linearization point is placed between its B_T and E_T events (that define its lifetime). Therefore, if $T1 \rightarrow_{\rho(TR)} T2$ ($T1$ ends before $T2$ begins), then $T1 \rightarrow_{ST} T2$. □

Let $\mathit{finite}(T, t)$ be the predicate "at time t, $MAX\_DATE_T \neq +\infty$".

Lemma 3. $\mathit{finite}(T, t) \Rightarrow \ell_T <_H t$.

Proof. The proof of the lemma consists in showing that the linearization point of a transaction T cannot be after the time at which MAX_DATE_T is assigned a finite value. There are three cases.

– By construction, if T aborts, its linearization point ℓ_T is the time at which the control variable MAX_DATE_T is assigned a finite value, which proves the lemma.
– If T is read-only and commits, again by construction, its linearization point ℓ_T is placed at the latest at the time at which MAX_DATE_T is assigned a finite value (if it ever is), which again proves the lemma.
– If T writes and commits, ℓ_T is placed during its try_to_commit() operation, while T holds the locks of every object that it has read. (If MAX_DATE_T had a finite value before it acquired all the locks, it would not commit due to line 15.) Let us notice that MAX_DATE_T can be assigned a finite value only by an update transaction holding a lock on a base object previously read by T. As T releases the locks just before committing (line 20), it follows that ℓ_T occurs before the time at which MAX_DATE_T is assigned a finite value, which proves the last case of the lemma. □

Let $rs_X(T, t)$ be the predicate "at time t, $T \in RS_X$ or $MAX\_DATE_T \neq +\infty$ ".

In the following, $AL_T(X, op)$ denotes the event associated with the acquisition of the lock on the object X issued by the transaction T during an invocation of op where op is $X.\mathsf{read}_T()$ or $\mathsf{try\_to\_commit}_T()$.

Similarly, $RL_T(X, op)$ denotes the event associated with the release of the lock on the object X issued by the transaction T during an invocation of op. Let us recall that, as $<_H$ (the shared memory history) is a total order, each event in H (including now $AL_T(X, op)$ and $RL_T(X, op)$) can be seen as a date of the time line. This "date" view of a sequential history on events will be used in the following proofs.

Lemma 4. $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow \nexists T'_W$ *such that* $\big(T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R\big) \wedge \big(w_{T'_W}(X) \in H\big)$.

Proof. The proof is by contradiction. Let us assume that there are transactions T_W, T'_W and T_R and an object X such that (1) $T_W \xrightarrow{X}_{rf} T_R$, (2) $w_{T'_W}(X)v' \in H$ and (3) $T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R$.

As both T_W and T'_W write X (shared memory accesses), they have necessarily committed (a write in shared memory occurs only at line 19 during the execution of try_to_commit(), abbreviated ttc in the following). Moreover, their linearization points ℓ_{T_W} and $\ell_{T'_W}$ occur while they hold the lock on X (before committing), from which we have the following implications:

$$\begin{aligned}
T_W \rightarrow_{ST} T'_W &\Leftrightarrow \ell_{T_W} <_H \ell_{T'_W},\\
\ell_{T_W} <_H \ell_{T'_W} &\Rightarrow RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T'_W}(X, \mathsf{ttc}),\\
RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T'_W}(X, \mathsf{ttc}) &\Rightarrow w_{T_W}(X)v <_H w_{T'_W}(X)v',\\
\big(T_W \xrightarrow{X}_{rf} T_R\big) \wedge \big(w_{T_W}(X)v <_H w_{T'_W}(X)\, v'\big) &\Rightarrow\\
&\quad w_{T_W}(X)\, v <_H r_{T_R}(X)v <_H w_{T'_W}(X)v'.
\end{aligned}$$

Hence, we have $(T_W \rightarrow_{ST} T'_W) \Rightarrow (r_{T_R}(X)v <_H w_{T'_W}(X)v')$.

On another side, a transaction T that reads an object X always adds its id to RS_X before releasing the lock on X. Therefore, the predicate $rs_X(T, RL_T(X, X.\mathsf{read}_T()))$ is true (a transaction T is removed from RS_X only after MAX_DATE_T has been assigned a finite value). From this observation and the previous result, we have: $r_{T_R}(X)v <_H w_{T'_W}(X)v' \wedge rs_X(T_R, RL_{T_R}(X, X.\mathsf{read}_{T_R}())) \Rightarrow rs_X(T_R, AL_{T'_W}(X, \mathsf{ttc}))$, and then

$$\begin{aligned}
\text{(Due to line 17)}\quad rs_X(T_R, AL_{T'_W}(X, \mathsf{ttc})) \wedge \big(w_{T'_W}(X)v' \in H\big) &\Rightarrow \mathit{finite}(T_R, \ell_{T'_W}),\\
\text{(Due to Lemma 3)}\quad \mathit{finite}(T_R, \ell_{T'_W}) &\Rightarrow \ell_{T_R} <_H \ell_{T'_W},\\
\text{(and finally)}\quad \ell_{T_R} <_H \ell_{T'_W} &\Leftrightarrow T_R \rightarrow_{ST} T'_W,
\end{aligned}$$

which proves that, contrarily to the initial assumption, T'_W cannot precede T_R in the sequential transaction history $\widehat{ST}$. □

Lemma 5. $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow (T_W \rightarrow_{ST} T_R)$.

Proof. The proof is made up of two parts. First it is shown that $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow \neg\mathit{finite}(T_R, \ell_{T_W})$, and then it is shown that $\neg\mathit{finite}(T_R, \ell_{T_W}) \wedge T_W \xrightarrow{X}_{rf} T_R \Rightarrow (T_W \rightarrow_{ST} T_R)$.

Part 1: Proof of $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow \neg\mathit{finite}(T_R, \ell_{T_W})$.

Let us assume by contradiction that $\mathit{finite}(T_R, \ell_{T_W})$ is true. Due to the atomic $C\&S()$ operation used at line 17, $MAX\_DATE_{T_R}$ is assigned a finite value only once. $MAX\_DATE_{T_R}$ will then be strictly smaller than the value of $X.date$ after T_W writes it. The test at line 05 of the $X.\mathsf{read}_T()$ operation will then fail, leading to $\neg(T_W \xrightarrow{X}_{rf} T_R)$. Summarizing this reasoning, we have $\mathit{finite}(T_R, \ell_{T_W}) \Rightarrow \neg(T_W \xrightarrow{X}_{rf} T_R)$, whose contrapositive is what we wanted to prove.

Part 2: Proof of $\neg\mathit{finite}(T_R, \ell_{T_W}) \wedge T_W \xrightarrow{X}_{rf} T_R \Rightarrow (T_W \rightarrow_{ST} T_R)$.

As defined earlier, the linearization point ℓ_{T_R} depends on the fact that T_R commits or aborts, and is a read-only or update transaction. The proof considers the three possible cases.

– If T_R is an update transaction that commits, its linearization point ℓ_{T_R} occurs after its invocation of $\mathsf{try\_to\_commit}()$. Due to this observation, the fact that T_W releases its locks after its linearization point, and $T_W \xrightarrow{X}_{rf} T_R$, we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$.
– If T_R is a (read-only or update) transaction that aborts, its linearization point ℓ_{T_R} is the time at which $MAX\_DATE_{T_R}$ is assigned a finite value. Because $T_W \xrightarrow{X}_{rf} T_R$ we have $\neg\mathit{finite}(T_R, \ell_{T_W})$. Moreover, due to $\neg\mathit{finite}(T_R, \ell_{T_W})$ and the fact that T_R aborts, we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$. It follows that $T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.
– If T_R is a read-only transaction that commits, its linearization point ℓ_{T_R} is placed either at the time at which $MAX\_DATE_{T_R}$ is assigned a finite value (then the case is the same as a transaction that aborts, see before), or at the time of the test during its last read operation (line 05). In the latter case, we have $w_{T_W}(X)v <_H \ell_{T_W} <_H RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T_R}(X, X.\mathsf{read}_{T_R}()) <_H r_{T_R}(X)v <_H \ell_{T_R}$, from which we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$.

Hence, in all cases, we have $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow (T_W \rightarrow_{ST} T_R)$. □

Theorem 1. *Every transaction history produced by the algorithm described in Figure 2 satisfies the opacity consistency property.*

Proof. The proof follows from the construction of the set $\rho(TR)$ (Section 4.3, Section 4.4, and text of the algorithm), the definition of the linearization points (Section 4.4), and the Lemmas 1, 2, 4 and 5. □

5 Conclusion

The aim of this paper was to show that Software Transactional Memory is a novel promising approach to address multiprocess programming. It discharges the programmer from using and managing base synchronization mechanism. The programmer only has to focus his/her attention (1) on the decomposition of his/her application into processes, and, for each process, (2) on its decomposition into atomic units.

To illustrate these notions, two STM protocols have been presented. Both are based on a logical global clock, and on locks associated with each shared object. The second protocol has been proved correct. Such a proof constitutes a step in establishing the foundations of STM systems.

References

1. Attiya, H.: Needed: Foundations for Transactional Memory. ACM Sigact News, Distributed Computing Column 39(1), 59–61 (2008)
2. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
3. Felber, P., Fetzer, Ch., Guerraoui, R., Harris, T.: Transactions are coming Back, but Are They The Same? ACM Sigact News, Distributed Computing Column 39(1), 48–58 (2008)
4. Guerraoui, R., Kapałka, M.: On the Correctness of Transactional Memory. In: Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Par. Progr. (PPoPP 2008), pp. 175–184 (2008)
5. Harris, T., Cristal, A., Unsal, O.S., Ayguade, E., Gagliardi, F., Smith, B., Valero, M.: Transactional Memory: an Overview. IEEE Micro 27(3), 8–29 (2007)
6. Herlihy, M.P., Luchangco, V.: Distributed Computing and the Multicore Revolution. ACM SIGACT News 39(1), 62–72 (2008)
7. Herlihy, M.P., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-free Data Structures. In: Proc. 20th ACM Int'l Symp. on Comp. Arch (ISCA 1993), pp. 289–300 (1993)
8. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
9. Imbs, D., Raynal, M.: Provable STM Properties: Leveraging Clock and Locks to Favor Commit and Early Abort. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 67–78. Springer, Heidelberg (2008)
10. Imbs, D., Raynal, M.: On the Consistency Conditions of Transactional Memories. Tech Report #1917, 23 pages, IRISA, Université de Rennes, France (2009)
11. Imbs, D., Raynal, M.: A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition. In: 16th Colloquium on Structural Information and Communication Complexity (SIROCCO 2009). LNCS. Springer, Heidelberg (2009)
12. Larus, J., Kozyrakis, Ch.: Transactional Memory: Is TM the Answer for Improving Parallel Programming? Communications of the ACM 51(7), 80–89 (2008)
13. Papadimitriou, Ch.H.: The Serializability of Concurrent Updates. Journal of the ACM 26(4), 631–653 (1979)
14. Raynal, M.: Synchronization is coming back, but is it the same? Keynote Speech. In: IEEE 22nd Int'l Conf. on Advanced Inf. Networking and Applications (AINA 2008), pp. 1–10 (2008)
15. Shavit, N., Touitou, D.: Software Transactional Memory. Distributed Computing 10(2), 99–116 (1997)

Sparse Matrix Operations on Multi-core Architectures

Carsten Trinitis[1], Tilman Küstner[1], Josef Weidendorfer[1], and Jasmin Smajic[2]

[1] Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik
Technische Universität München, Germany
{Carsten.Trinitis,Tilman.Kuestner,Josef.Weidendorfer}@in.tum.de
[2] ABB Corporate Research Center
Baden-Daettwil, Switzerland
Jasmin.Smajic@ch.abb.com

Abstract. This paper compares various contemporary multi-core based microprocessor architectures with different memory interconnects regarding performance, speedup, and parallel efficiency. Sparse matrix operations are used as a benchmark application from the area of electrical engineering. Within this context, thread to core pinnning and cache optimization are two important aspects which are investigated in more detail.

Keywords: Multi-core, pinning, cache optimization, performance optimization, sparse matrices.

1 Introduction

Sparse matrix operations can be among of the most difficult applications in numerical simulation. Within this research field, an application from electrical engineering has been analyzed, utilizing various tools developed at Technische Universität München, namely within the Munich Multicore Initiative (MMI) [1]. An OpenMP [6] based parallel version of the code was investigated with regard to possible performance improvements using MMI's tools.

For NDA reasons with the project partner, all simulations were conducted with synthetic model data.

Within recent years, a trend towards multi-core architectures with currently four cores for a standard x86 based architecture can be observed. In order to fully utilize the potential of such machines, applications must be parallelized and analyzed with regard to runtime, speedup, and parallel efficiency. With multi-core architectures becoming more and more complex, it is of crucial importance to compare different hardware architectures with regard to how well they are suited for the application under investigation. Thus, six contemporary x86 based architectures have been investigated for a sparse matrix solver. The remainder

[1] http://mmi.in.tum.de

of this paper is organized as follows: Section 2 will give a brief introduction to sparse matrix problems, section 3 gives an overview on the hardware which was used for benchmarking, and section 4 presents and analyzes runtimes, speedup, and parallel efficiency obtained on these systems. Section 5 provides an analysis on the program's cache behavior, and section 6 concludes and gives an outlook on future work.

2 Sparse Matrix Operations

Making a sparse matrix application scale is a fairly difficult task. In the case discussed in this paper, less than one per cent of the entries in the matrix are non-zero, and distributed according to a pattern given by the electrical engineering application. This makes it difficult to access the entries in the calculations in a way that is cache friendly. The first code developed for this particular application was developed in the mid eighties [8], when the primary focus in the code structure was to minimize the matrix bandwidth (and thus minimizing the number of floating point operations), i.e. keep the number of fill-ins to a minimum [2].

To minimize the number of floating point operations, i.e. keep the number of fill-ins to a minimum ("fill-ins" means additional matrix entries created during the Gauss elimination or LDU-factorization that were zero in the original matrix). While it is still important to avoid unnecessary floating point operations, memory performance has improved in a much slower pace than CPU-performance, compared to when the initial code was written for this application. That means that the number of floating point operations may no longer be the biggest bottleneck when it comes to application scalability.

This application uses the Markowitz criteria [4] to minimize the number of fill-ins during the LDU-factorization [8]. It performs a large number of sparse matrix operations that are totally independent of each other, i.e. it is not the LDU-operations that have been parallelized - it is the independent calculations that execute in parallel.

3 Hardware Environment

Comprehensive benchmark tests were carried out on six different systems, with different hardware architectures. In the list below, the nickname of the system, followed by the processor type and amount of main memory are given.

– *Nehalem* – 2×Intel Xeon X5570 (*Gainestown*, quad-core (8 cores with HyperThreading enabled), 2.93 GHz, 8 MB shared L3 cache, 2×QPI), 12 GB DDR3 RAM
– *Dunnington* – 4× Intel Xeon X7460 (*Dunnington*, hexa-core, 2.66 GHz, 16 MB shared L3 cache), 1066 MHz FSB, 32 GB DDR2 RAM
– *Shanghai* – 2×AMD Opteron 2376 (*Shanghai*, quad-core, 2.41 GHz, 6 MB shared L3 cache, 2×HyperTransport 3.0), 32 GB DDR3 RAM

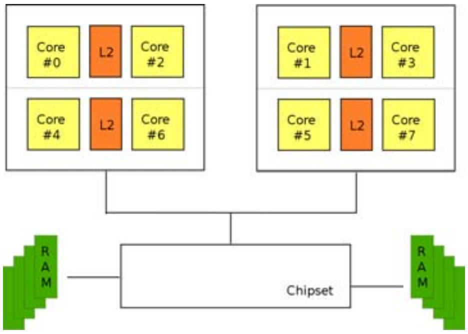


Fig. 1. Front Side Bus based system

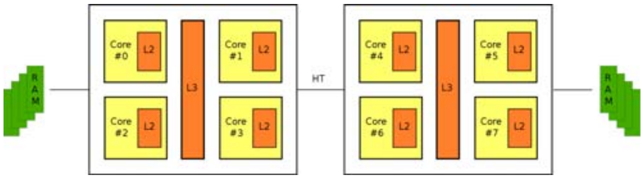


Fig. 2. NUMA like system

– *Barcelona* – 2×AMD Opteron 2352 (*Barcelona*, quad-core, 2.11 GHz, 2 MB shared L3 cache, 2×HyperTransport 3.0), 16 GB DDR2 RAM
– *X4600* – Sun Fire X4600 M2: 8×AMD Opteron 8218 (*Santa Rosa*, 2.60 GHz, dual-core, 1 MB L3 cache per core, 3×HyperTransport), 64 GB DDR2 RAM
– *Clovertown* – 2×Intel Xeon X5355 (*Clovertown*, quad-core, 2.66 GHz, 4 MB L2 cache shared across two cores), 1333 MHz FSB, 8 GB DDR2 RAM

As can be seen from this list, the processor architectures comprise the latest Intel architecture codenamed "Nehalem" as well the latest AMD processor "Shanghai" and some "older" processor types by both Intel and AMD.

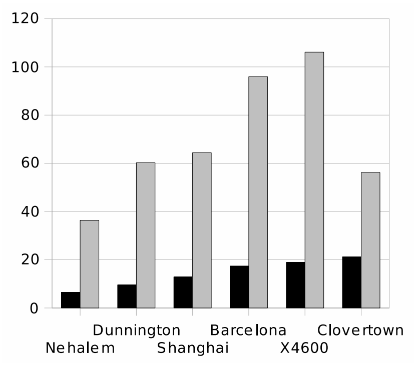
These systems represent two different architectures:

– A front side bus (FSB) based system, as depicted in figure 1, represented by the *Clovertown* and *Dunnington* systems.
– A NUMA like system, as depicted in figure 2, represented by the *Nehalem*, *Barcelona*, *Shanghai*, and *x4600* systems.

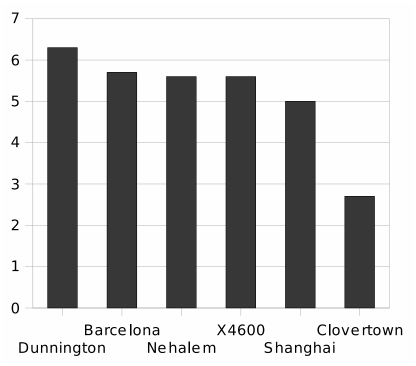
4 Measurements

As reported in previous research work carried out by the Munich Multicore Initiative at LRR-TUM, thread to core pinning does have a non negligible impact on parallel program performance on multicore architectures. The optimal pinnning can vary significantly, depending on the processor, on the overall system architecture as well as on the cache hierarchy [7], [5]. With the `autopin` tool developed by MMI, thread to core pinning on all available architectures was thoroughly tested for our sparse matrix operations. To pin a thread to a specific processor core, `autopin` makes use of the system call `sched_setaffinity`. This prevents threads from moving between cores, which would result in poor cache usage. In some cases it is desirable to not use all cores on a chip, i.e. in order to avoid pinning to cores which share the same cache or which are located on the same chip. For details on cache usage see section 5.

The first set of measurements focused on total program runtime. These were carried out on all six architectures with eight parallel threads. The pinning order used here was the optimal pinning order as determined by `autopin`, i.e. `0, 4, 1, 5, 2, 6, 3, 7`, with the core numbers denoting the cores as depicted in figures 1 and 2. As can be seen from figure 3a, Intel's Nehalem architecture shows the best performance, followed by AMD Shanghai and Barcelona. Here, all available cores were utilized, i.e. one thread was pinned to each core. This also applies to the Clovertown system. For the Dunnington and X4600 systems (with 24 and 16 available cores, respectively), the optimal pinning for eight threads was determined with MMI's `autopin` tool. This turned out to be using one core of the dual core chips for the X4600 architecture, and two cores of the hexacore chips on the Dunnington architecture, such that they do not share a common L2 cache.



(a) Total time (in seconds) with one (gray) and eight (black) threads



(b) Speedup with eight threads

Fig. 3. Total time (a) and speedup (b)

Next, parallel efficiency was investigated by determining the speedup factors on all six architectures with the same pinning. For eight threads, most systems showed an average speedup of 5.0 to 5.5. The poorest parallel efficiency was measured on the Clovertown system with a speedup of only 2.6, whereas Dunnington system performed best with a speedup of 6.3. This good performance is attributed to its large 16 MB last-level cache. It must be noted, however, that only 8 threads were run on a 4 × 6 core system, i.e. one third of the cores was utilized on each socket, allowing 8MB cache per core. When pinning a thread to all 24 cores, the efficiency drops down to 16%, which is due to the high load on the memory link. Hence, it could be found out with `autopin` that the application scales optimal on the Dunnington system when utilizing two cores per socket. Moreover, this also refers to the overall runtime: On the 24-core Dunnington system, optimal performance (i.e. runtime) was achieved with 8 threads at 2 threads per core. Therefore, it is advisable to investigate the target architecture an application is supposed to run on with regard to optimal pinning before utilizing all available cores, as additional cores do not deliver additional performance in certain cases.

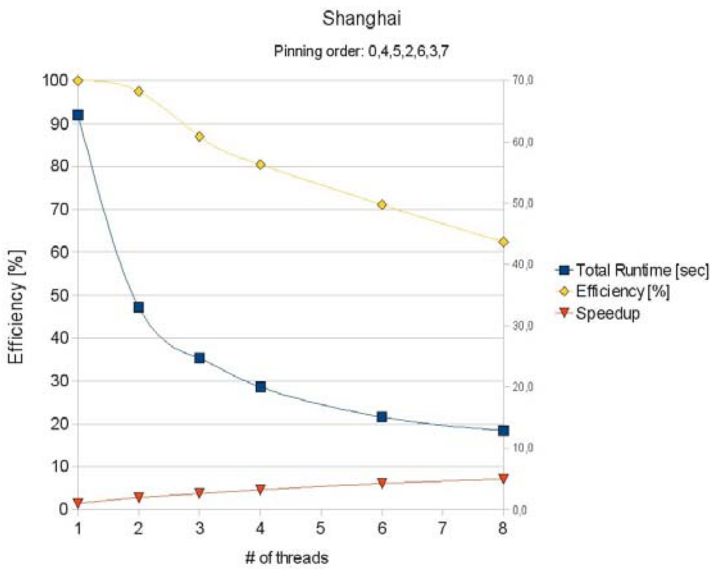


Fig. 4. Total runtime, parallel efficiency, and speedup for Shanghai

Figure 3 depicts the total runtime for one and eight threads as well as the speedup for eight threads on all six architectures. Figures 4 and 5 compare total runtime, speedup, and parallel efficiency for the Shanghai and Nehalem systems in more detail. In these figures, the respective optimal pinning, starting from 1 core, has been used. The pinning order is depicted in the diagrams. Thus, e.g. for the Shanghai (see fig. 4, 1 thread was pinned to core #0, 2 threads were pinned to cores #0 and #4, 3 threads to cores #0, #4, and #5, etc. . For these two latest AMD and Intel systems, parallel efficiency is above 90 per cent for two cores and above 80 per cent for up to four cores, which is due to the fact that threads are always pinned to cores in such a way that the available cache is used optimally.

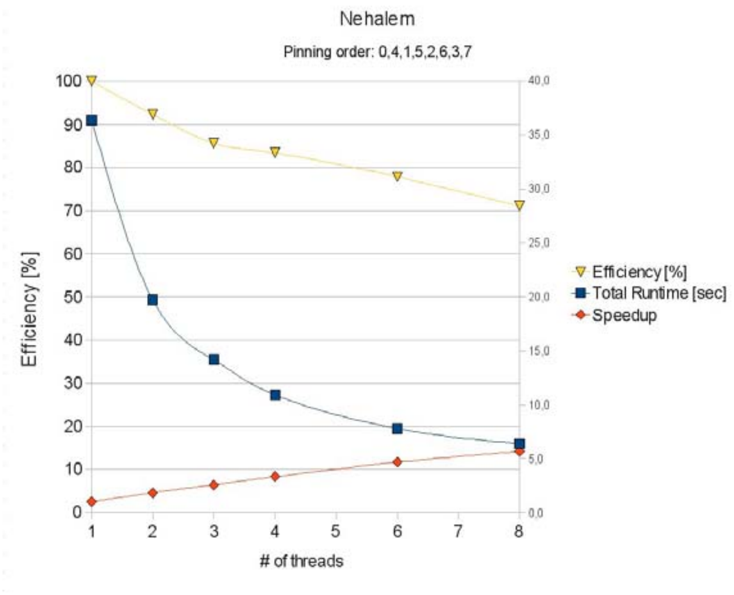


Fig. 5. Total runtime, parallel efficiency, and speedup for Nehalem

5 Analysis of Cache Behavior

In addition to the measurements presented in the previous section, the application's cache behavior was analyzed in order to spot further bottlenecks and obtain possible performance improvements. For simulating the application's cache usage, the tools Callgrind/KCachegrind [10],[9], which were developed in the DIME [1] project, and extended by MMI, were used for the investigations. Callgrind is part of the open-source project Valgrind [3], which consists of tools for correctness checking and profiling built on a infrastructure for dynamic runtime instrumentation.

Three test cases were examined:

– A single thread running on a core with 4 MB L2 cache.
– Four threads running on a quad-core processor with 4 MB shared L2 cache (as on the systems investigated in previous sections).
– Four threads running on a quad core processor with 16 MB shared L2 cache.

The main objective for these test cases was to determine data sharing characteristics of the given OpenMP parallelization. The test cases approximate the behavior on multi-core architectures with shared last-level caches realistically. The main difference between reality and the simulations carried out with MMI's cache analysis tools is that a shared last-level cache is normally realized as a third cache level, but this has no impact on the simulation results.

The simulation showed almost exactly the same number of instruction fetches and data references in all test cases, when aggregated over all threads. This

Table 1. Simulated L2 cache misses

	1 thread, 4 MB L2	4 threads, 4 MB L2	4 threads, 16 MB L2
Total	987,490,834	1,380,396,029	57,413,638
`airflowb`	174,203,730	265,674,181	285,145
`lqdflowb`	184,610,326	264,806,996	230,186

comes as no surprise as the same input data was used in all cases, but it also proofs the comparability of the L2 cache misses, which are displayed in table 1.

Taking a look at the first row of the table above denoting the cache misses of the entire program, it can be noticed that there is a major drop when moving to larger cache size (i.e. 16MB). This means that this amount of cache is capable of holding the input data of the examined bus model. This also correlates to the good performance numbers on the Dunnington system. When moving from one to four threads sharing the smaller 4 MB cache, the increase in cache misses is not as tremendous. This is because a considerable amount of data can be shared across the threads. Also, the data shows that the parallelization does not increase the total memory space requirement in contrast to the sequential version. Clearly, shared caches are beneficial for the given application.

Regarding potential performance bottlenecks in the code, the functions `airflowb` and `lqdflowb` were identified as the ones which caused the most cache misses. In test cases one and two (small L2 cache) these two function contribute to nearly a quarter (22% to 24%) of all cache misses. In simulations carried out for 16 MB cache, however, they only account for 5.6% of the total misses.

6 Conclusions and Future Work

Six contemporary multicore architectures were compared with a parallel reference application for sparse matrix solvers. The application was parallelized with a shared memory model under OpenMP. Thread to core pinning and cache otimization were investigated with regard to the application under consideration. The investigations showed, that, depending on the processor architecture as well as on the memory interconnect, it is not always advisable to utilize all available core in a system. With regard to cache optimization, is has been shown that potential bottlenecks can be easily detected with MMI's simulation tools.

References

1. DiME DFG Project, Web Page, `http://www10.informatik.uni-erlangen.de/Research/Projects/DiME`
2. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: ACM Annual Conference/Annual Meeting, Proceedings of the 24th national conference, pp. 157–172 (1969)
3. The Valgrind Developers. Valgrind Web Page, `http://valgrind.org/`

4. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, Oxford (1986)
5. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: autopin - automated optimization of thread-to-core pinning on multicore system 3 (2008)
6. OpenMP.org. The OpenMP API specification for parallel programming, `http://www.openmp.org/`
7. Ott, M., Klug, T., Weidendorfer, J., Trinitis, C.: autopin - automated optimization of thread-to-core pinning on multicore systems. In: First Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG). Workshop proceedings, 1st Multiprog workshop, Gothenburg, Sweden (January 2008)
8. Tinney, W.F., Brandwajn, V., Chan, S.M.: Sparse vector methods. IEEE Transactions on Power Apparatus and Systems PAS-104(2) (February 1985)
9. Weidendorfer, J.: KCachegrind Web Page, `http://kcachegrind.sourceforge.net/`
10. Weidendorfer, J., Kowarschik, M., Trinitis, C.: A tool suite for simulation based analysis of memory access behavior. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3038, pp. 440–447. Springer, Heidelberg (2004)

Multi-granularity Parallel Computing in a Genome-Scale Molecular Evolution Application

Jesse D. Walters[2,4], Thomas B. Bair[2], Terry A. Braun[1,2,4,5], Todd E. Scheetz[1,2,4,5], John P. Robinson[1,4], and Thomas L. Casavant[1,2,3,4]

[1] Center for Bioinformatics and Computational Biology
[2] Coordinated Laboratory for Computational Genomics
[3] Department of Biomedical Engineering
[4] Department of Electrical and Computer Engineering
[5] Department of Ophthalmology and Visual Sciences,
University of Iowa, Iowa City, IA 52242 USA
clcg@eng.uiowa.edu

Abstract. Previously [1], we reported a coarse-grained parallel computational approach to identifying rare molecular evolutionary events often referred to as horizontal gene transfers. Very high degrees of parallelism (up to 65x speedup on 4,096 processors) were reported, yet the overall execution time for a realistic problem size was still on the order of 12 days. With the availability of large numbers of compute clusters, as well as genomic sequence from more than 2,000 species containing as many as 35,000 genes each, and trillions of sequence nucleotides in all, we demonstrated the computational feasibility of a method to examine "clusters" of genes using phylogenetic tree similarity as a distance metric. A full serial solution to this problem requires years of CPU time, yet only makes modest IPC and memory demands; thus, it is an ideal candidate for a grid computing approach involving low-cost compute nodes. This paper now describes a multiple granularity parallelism solution that includes exploitation of multi-core shared memory nodes to address fine-grained aspects in the tree-clustering phase of our previous deployment of XenoCluster 1.0. In addition to benchmarking results that show up to 80% speedup efficiency on 8 CPU cores, we report on the biological accuracy and relevance of our results compared to a reported set of known xenologs in yeast.

1 Introduction

Historically, evolutionary biological research has proceeded from painstaking taxonomical classification according to the physical characteristics of hundreds of organisms and species. The Human Genome Project and subsequent application of massive DNA and RNA sequencing capacity has made available rich datasets capable of answering broad evolutionary questions from a molecular point of view. These trillions of nucleotides of DNA sequence data from thousands of species containing as many as 35,000 genes each makes it possible to pose biological and biomedical questions that just a few years ago would have been inconceivable. In our previous paper [1], we described XenoCluster 1.0 which addressed one such setting known as horizontal gene transfer, or jumping genes. In addition, general issues facing the use of heterogeneous networks for such problems were addressed. The results of that initial implementation showed that it was possible to efficiently harness more than 4,096 processors organized in a heterogeneous grid of modest sized compute clusters to reduce the overall computation time of a typical problem setting from more than 2 years to roughly 12 days. This paper extends that initial work to exploit fine-grained shared memory parallelization of previously unparallelized portions of that application which further reduces practical execution times by another order of magnitude. We also address the issue of biological accuracy and relevance of predictions using recently reported xenologs in S.cerevisiae (budding yeast). kground and Related Work

For the sake of completeness, an abbreviated overview of some background information as well as our general discovery method [1] is presented here.

2 Biological Background

Typical genes are transferred through lineages, from one generation to the next within a species. However, an alternate form of "inheritance" is possible in which genetic material crosses species boundaries. This form of inheritance is termed horizontal gene transfer. Our ability to identify patterns of horizontal gene transfer can increase our understanding of the evolution of species and the structure of the tree of life.

General features of horizontal gene transfer include higher inter-species sequence similarity between two taxa (species) that are in different clades (branches) of the consensus tree. To accomplish this, a broad set of species must be sampled. The limiting factors in this process are the availability of sequence for a large number of taxa and the capability to harness sufficient computational power to identify orthologous sequences, construct phylogenetic trees for each orthologous set of genes, and then compare the trees derived from each orthologous set to identify non-evolutionary inheritance patterns.

2.1 Computational Background

Several operations are necessary to identify horizontal gene transfers, or other anomalous gene inheritance events. First, orthologous sequences must be identified. Next these sequences must be aligned, and phylogenetic trees created. Finally, the tree structures must be compared to identify atypical patterns of inheritance.

There are several computational methods for determining orthology. The most commonly accepted method is based upon a strongly connected graph constructed of nodes representing genes across species in which each element is more similar to the others in the set than to genes from outside the set. There are several previously generated set of orthologs publicly available. These include COG and KOG [2], OrthoMCL [3] and EGO [4]. The method employed in our work incorporates sequences from all organisms available in NCBI's non-redundant amino acid database (NR) ad uses a straight-forward reciprocal BLAST criteria.

Phylogenetic analysis allows determination of the most likely pattern of inheritance of a gene. Programs such as PHYLIP [5] and PAUP [6] are commonly used to construct phylogenetic trees, based upon an aligned set of orthologous sequences. High-performance parallel phylogenetic inference has been extensively studied by Stamatakis in the RAxML-VI-HPC with impressive performance optimizations [7]. Due to its widespread acceptance and usage, we use PHYLIP in our work.

3 Methods and Solution

We now summarize our parallel grid of clusters solution in heterogeneous-latency networks. The algorithm is divided into 3 major phases:

1. Identification of a maximal set of orthologous genes.
2. Generation of phylogenetic trees resulting from orthologous groups.
3. Clustering of these trees into groups corresponding to genes which show consistent evolutionary behavior.

In phase 1, it is necessary to identify potential homologous genes for every gene in the union of a complex set of 1000s of species. This is accomplished by BLASTing [8] each gene against the set of all known genes in all species, and then performing a reciprocal BLAST operation to verify that the best hit for each gene hits the original gene with the highest rank score. This becomes the base set of orthologous gene groups to be used in phase 2 among which xenologs may be identified. The second phase involves the sequence trimming and multiple alignment of all members of each of the orthologous gene groups, followed by the automated generation of a phylogenetic tree for each aligned group. The final phase performs an all-pairs distance analysis of phylogenetic trees for all gene groups, and then uses a clustering technique to identify maximal sets of trees, which represent sets of genes which share a common evolutionary history. Design details of the procedure outlined above are presented in our previous paper [1], and are illustrated summarily in Figure 1. The highlights are repeated here for completeness only.

Ortholog identification was performed using the COE (Computational Orthology Engine) system, developed at the University of Iowa. COE identifies orthologous sequence groups using a reciprocal best-alignment strategy. Each mRNA RefSeq [9] sequence for a base species was BLASTed against NCBI's non-redundant amino acid database. For each BLAST result, the top hit of each species was selected, if and only if it met a stringent quality threshold criterion. If the threshold criterion was met, a reciprocal BLAST was performed with these top species hits against the RefSeq database [9] to further support the orthology inference.

Text parsing of the BLAST results was performed using custom BioPerl [10] scripts, and batch scheduling of all BLAST operations in phase 1 was performed using the Portable Batch System [11].

Once ortholog identification has been performed, the next phase is trimming, alignment, and phylogenetic tree generation. The trimming of sequences is done using a custom Perl script. Multiple sequence alignment is accomplished using the well-established clustalw software [12]. The final step in this phase is the generation of the

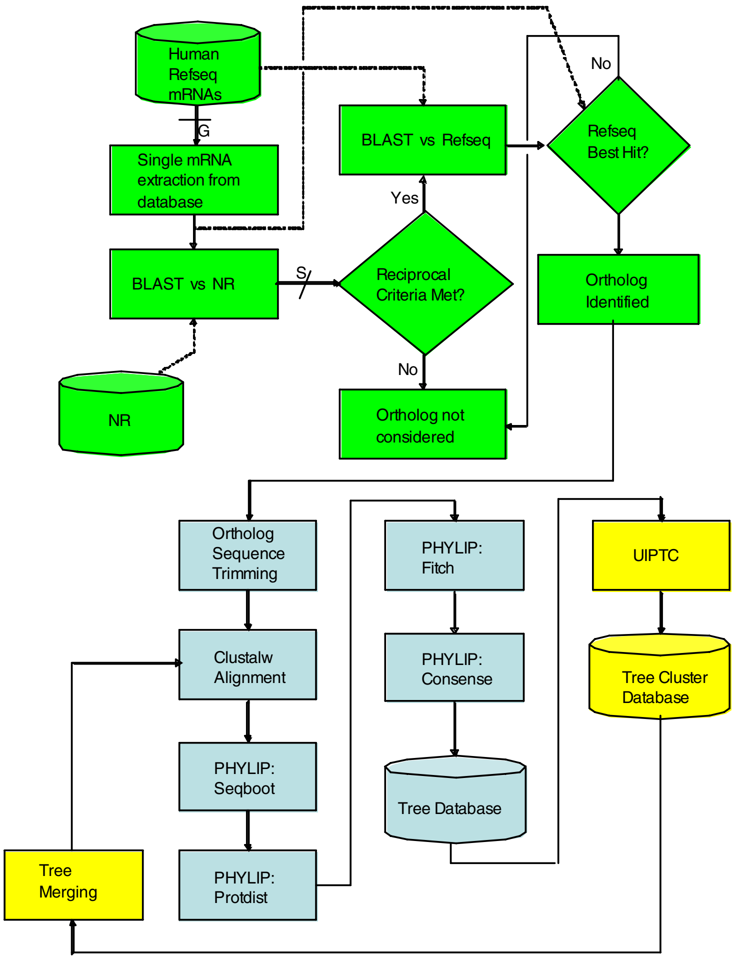


Fig. 1. A detailed flowchart of the XenoCluster approach

phylogenetic trees – the PHYLIP [5] software suite was used to generate each of the trees. These programs generated the sequence distance matrices and the phylogenetic trees for each of the bootstrap replicates. Finally, the consense program was run to obtain the consensus phylogenetic tree based upon the bootstrap replicates.

Finally, the phylogenetic tree clustering was performed from the results of the PHYLIP software package. This involves two main sub-phases – distance matrix generation, and clustering. Inter-tree distance was calculated using a modified version of the TreeRank [13] algorithm. Our adaptation of this algorithm was implemented in POSIX C with pthread support. Development of this software was done on Fedora 9.0 and Redhat Enterprise machines. The second sub-phase involved clustering, given a complete distance matrix from every tree to every other tree.

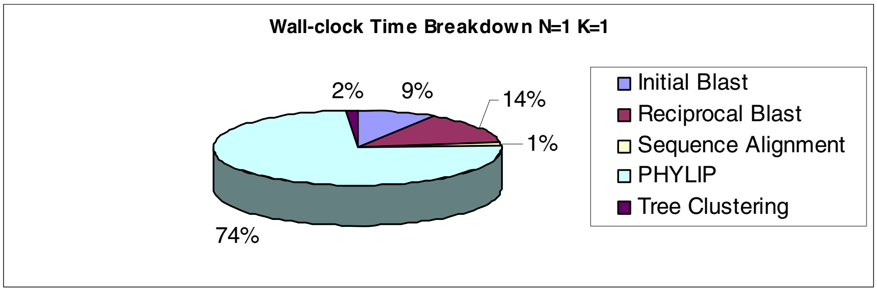
3.1 Grid/Cluster Implementation and Benchmarking Details

Each of the phases described above were implemented in a LINUX environment (2.2GHz dual Athlon with 2GB RAM running Fedora Redhat 9.0), and benchmark executions were performed using the largest set of human genes in April of 2005. For this analysis, and all benchmarks, this consisted of the set of all 20,364 known human RefSeq mRNA sequences. Runtime estimates for the first phase of the computation, which involved the COE system, varied significantly with system threshold parameters. The initial iteration of the system yielded an average of 588 cpu seconds per RefSeq mRNA. Variations of the aforementioned match length, alignment score and e-value thresholds can change the number of reciprocal BLASTs performed and therefore the average runtime. Thus, the values reported in Table 1 are an average taken across the entire set of 20,364 genes. The COE results yielded an average of 12.6 orthologs per human RefSeq mRNA. More relevant to performance, an average of 39 reciprocal BLASTs were performed for each RefSeq mRNA. Thus, approximately 32% of the reciprocal alignments were considered "true" orthologs by our method.

In our previous work [1], we discussed in detail the effects of deployment of XenoCluster on a large-scale grid of compute clusters. We utilized the figure of 588 seconds to estimate and bound the runtime of the entire dataset through the COE system. To confirm the accuracy of our execution-time predictive model, 20,364 mRNAs at an average of 588 seconds would yield 3,326 CPU hours of compute-time.

Table 1. Benchmark timings on 20,364 human genes for the component phases of XenoCluster run with 1 dual CPU node (cluster size 1). Timings taken on a 2.2GHz dual Athlon with 2GB RAM running Fedora Redhat 9.0.

Phase/component	**Time (Seconds)**	**# of Iterations**	**Total (Seconds)**
Intra Cluster IPC	124	1	124
Inter Cluster IPC	311	1	311
Initial BLAST	301	20,364	6,129,564
Reciprocal BLAST	12	794,196	9,530,352
Sequence Alignment	33	20,364	672,012
PHYLIP tree generation	2,518	20,364	51,276,552
Tree Clustering	1,036,800	1	1,036,800
Total	**1,040,099**	**855,291**	**68,645,715**
Days to completion			**794.5106**



A benchmark was then performed on a 16-node Linux cluster where the observed execution time was shortened to 207 cluster hours. This was very close to the expected time of approximately 12 days. Runtimes for PHYLIP were 579 cpu seconds at 100 bootstrap iterations, while an average of 2,518 cpu seconds was achieved at 500 bootstrap iterations. The tree clustering phase (UIPTC) results were extrapolated to reveal the estimated overall runtime of the full set of 20,364 genes.

Table 1 summarizes the detailed times (in wall-clock time units of seconds) of 5 computational and 2 communication elements of XenoCluster. Details are provided in our previous paper. Note that the first four computational elements parallelize cleanly across all genes. However, in the final phase, the times for tree clustering (UIPTC) do not show the effects of parallelization.

4 Results and Discussion

In [1] we presented the performance results of the coarse-grained grid-based parallelization of the approach outlined above. Now we present the results of fine-grained parallelization of the Tree Clustering (UIPTC) phase, as well as the biological validation of our method. In our benchmarking results shown above, UIPTC requires approximately 12.2 days of computational time. While this amounts to only 2% of the serial execution time as shown in Table 1, this phase now comprises 99% of the best-case parallel execution time as shown in Table 2.

Table 2. Execution times with a coarse-grained cluster-parallel implementation utilizing K=128 Linux Clusters, and N=32 CPUs/Cluster. A total of 4,096 processors in all for a net speedup of 65.

Phase/component	**Time (Seconds)**	**# of Iterations**	**Total (Seconds)**
Intra Cluster IPC	124	128	15,872
Inter Cluster IPC	311	32	9,952
Initial BLAST	301	5	1,505
Reciprocal BLAST	12	194	2,328
Sequence Alignment	33	5	165
PHYLIP tree generation	2,518	5	12,590
Tree Clustering	1,036,800	1	1,036,800
Total	1,040,099	242	1,079,212
Days to completion			**12.5**

The fine-grained parallelization of the UIPTC algorithm was done using the pthread multi-threading package for Linux 2.4 based kernels [14]. This package allows for several threads or light-weight processes to share memory but maintain independent execution paths. The most obvious place to harness multiprocessor capabilities in UIPTC was in the treeSim() function [1]. In this function, (NxN)/2 independent comparisons of the trees to one another is performed. These comparisons were randomly partitioned into sets. Each set is then run in a different thread.

The distribution of work to threads was accomplished by assigning a unique ID to each thread and providing the total number of threads in the process to each thread. The ID and number of threads allowed each thread to process a given row I in the NxN matrix and perform the following operation: mod(I, numThreads) == ID. If this statement was true, the thread takes local responsibility for all the computation in that row.

Table 3 shows the results of parallelization of UIPTC on a set of HP workstations with dual-core Opteron processors. For small numbers of CPUs, almost linear speedup is observed. The use of the rudimentary load balancing as mentioned earlier was adequate, and provided threads with desirable computational load. Although excellent speedup was achieved, most users will be limited to 2 or 4 CPUs per system simply because most x86 architectures do not scale well above 8 CPUs. Thus, 16 and 32 thread implementations would be unlikely to be efficient without additional message passing infrastructure [15]. Figure 2 and Table 3 show runtime speedup and efficiency results for UIPTC with different numbers of threads. Note that efficiency begins to decrease as the number of CPUs approaches 8. This can be attributed to memory latency and system bus bottlenecks. However, speedup is achieved as one increases the number of threads from 7 to 8, which means additional speedup could be achieved by simply using more CPUs. Note that memory usage was less than 10 megabytes and therefore not a limiting factor, however, memory demand grows at an O(n2) rate, which means it may be something to consider as datasets grow. Benchmarking was performed using HP workstation using 4 2.2 GHz Dual Core Opteron Processors. The benchmarking dataset was the complete yeast tree build described above.

4.1 Biological Validation and Interpretation of Results

Results for the COE phase of the XenoCluster system applied to the S.cerevisiae (yeast) species yielded on averaged 30 orthologs per yeast RefSeq gene. These orthologs were, on average, 361 amino acids long. A total of 128,190 orthologs were identified from 4,234 of 7,001 RefSeq genes. The 2,767 genes which did not yield orthologs, were most likely alternative transcripts which already were represented in the original gene set. This is a reasonable explanation as it is estimated that S.cerevisiae has approximately 5,000 genes. Of the 4,234 yeast gene ortholog sets identified above, 2,650 trees were produced from our PHYLIP pipeline. Approximately 2,000 trees were not created because they lacked the minimum criteria of 5 orthologous species. This criterion was utilized because trees with less than 5 species may mis-represent an HGT event.

Table 3. UIPTC parallel runtime and efficiency results for 1 through 8 CPUs

Number of CPUs	Observed Runtime	Theoretical Runtime	Efficiency%
1	7434	7434	100
2	3781	3717	98.27818133
3	2548	2478	97.17514124
4	1995	1858.5	92.65536723
5	1699	1486.8	85.72773742
6	1403	1239	86.76351897
7	1227	1062	84.46327684
8	1116	929.25	79.9031477

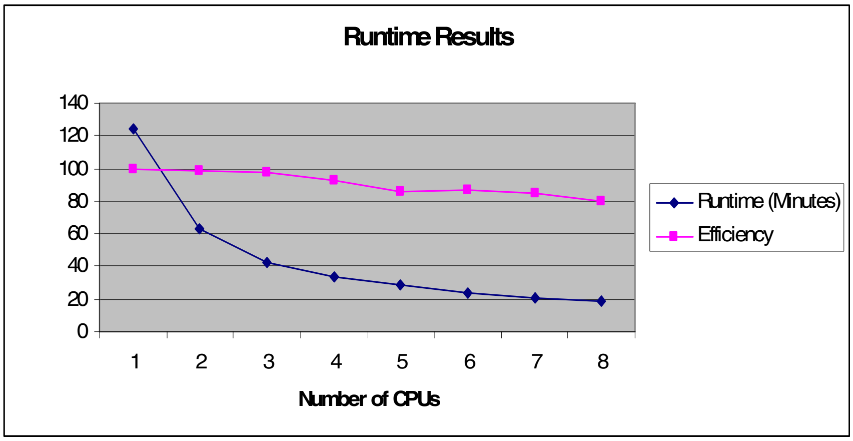


Fig. 2. UIPTC runtime and efficiency results from Table 3

Several iterations of phylogenetic clustering of the yeast genome were then performed utilizing several different similarity threshold values. These thresholds were the minimum TreeRank similarity needed to classify a tree in question, into a tree/gene cluster. Threshold values varied from 60 % up to 97 percent. Figure 3 shows the number of clusters formed for each of the threshold values used. Most of the clusters remain very small, while one or two large "consensus" clusters represent the majority of the sequences. For example, for the 95% identity clustering, the 87 resulting clusters consisted of 1 cluster of size 2129 trees, and 86 relatively small clusters consisting of fewer than 5 trees each.

To validate the utility of this approach, a list of 10 independently identified S.Cerevisiae horizontal gene transfer (HGT) cases were identified from the published in biological literature [16] and observed in the XenoCluster dataset. Each of the 10 RefSeq identifiers were BLASTed against the latest RefSeq release to identify their current RefSeq ids. We then examined the position of these genes in the resulting clustering. The results of this analysis are shown in Table 4. Four of the ten genes (YFR055W, YMR090W, YOL164W, YJL217W) lacked sufficient homology to other available species to meet our criteria of 5 species for reciprocal BLAST hits, and thus could neither be confirmed nor denied as valid xenologs. Of the remaining six, two were identified as belonging to singleton clusters (i.e., lacked similarity to the phylogenetic trees of any other yeast genes) at thresholds of 85% (YDR540C) and 90% (YJL218W). These two examples provide the strongest evidence of the validity of our approach. However, the remaining four cases (YPL245W, YKL216W, YNR057C, and YNR058W) were as yet unaccounted for.

In addition to a percent identity threshold, a secondary clustering criterion was implemented to improve HGT detection. The secondary criterion specifies a minimum number of trees that must meet the percent identity threshold in order for the tree in

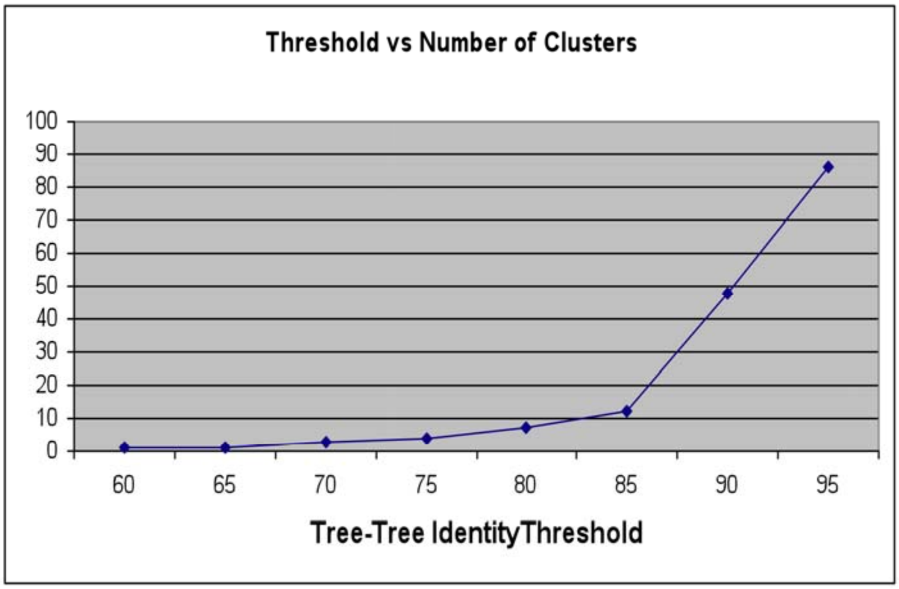


Fig. 3. Tree similarity threshold versus the number of resulting clusters. At very low thresholds, all genes/trees form a single cluster, while at high thresholds the number of clusters increases, yet most genes are found to be contained in a small number of relatively large clusters.

Table 4. Validation of Xenocluster using "known" yeast Xenologs [16]. %FS is the percentage to form a singleton cluster. %JL is the percentage to join the large cluster. #LL is the number of links to other trees in the largest cluster.

Yeast Gene ID	Gene name	%FS	%JL	# LL
YFR055W	Hypoth Protein			
YMR090W	Hypoth Protein			
YOL164W	BDS1			
YJL217W	Hypoth Protein			
YDR540C	Hypoth Protein	85%		
YJL218W	Hypoth Protein	90%		
YPL245W	Hypoth Protein		97%	25
YKL216W	URA1		97%	12
YNR057C	BIO4		97%	7
YNR058W	BIO3		97%	1

question to be incorporated into the cluster. Currently, if a single tree-tree comparison yields a result greater than or equal to the percent identity threshold, the tree in question will become part of that cluster. By applying this criteria to the single largest cluster (the one that represents the canonical structure the tree illustrating the true evolutionary position of yeast in the tree of life), we are able to distill trees (or genes) whose similarity to this dominant structure only consists of links to very few other members of that cluster. Our hypothesis was that these genes would also be excellent candidates as the result of GHT events.

With our current clustering results, we have observed several strongly related trees within the large consensus cluster. These trees may have as many as 1,005 tree-tree comparisons with other trees in the same cluster that are greater than or equal to the 95% percent identity threshold. However, many of the trees in this large cluster have only 1 tree-tree comparison that is equal to or greater than the selected percent identity threshold. By requiring more than a single tree-tree match greater than or equal to the specified percent identity, we were able to identify clusters that contain only trees that are strongly related to each other. Figure 4 displays in sorted order, the number of tree-tree comparisons from the consensus cluster that have greater than or equal to the 95 percent identity threshold. The four documented HGT cases that are in this cluster all have 25 or fewer tree-tree comparisons that meet the 95% identity threshold. The strongest of these is YNR058W, which shows 95% similarity to only one other gene in the large cluster. Thus, of the six known cases of HGT in yeast which have sufficient orthology information, all are known to be either very unique with respect to the phylogenetic tree structure, or to only have weak evidence for similarity to other genes.

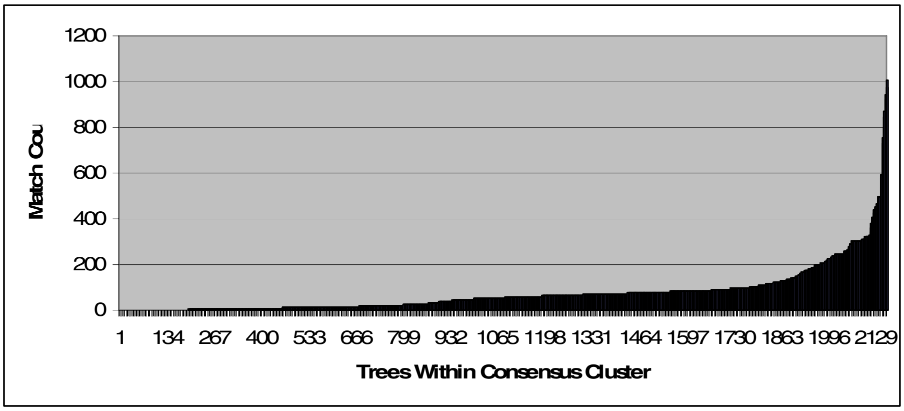


Fig. 4. Analysis of the number of tree-tree comparisons meeting the similarity criteria specified. The largest number of connections is a tree with similarity of over 97% to 1,004 other trees. There are approximately 150 trees with fewer than 25 links to other trees

References

1. Walters, J., Casavant, T., Robinson, J., Bair, T., Braun, T., Scheetz, T.: XenoCluster: A Grid Computing Approach to Finding Ancient Evolutionary Anomolies. In: Malyshkin, V.E. (ed.) PaCT 2005. LNCS, vol. 3606, pp. 355–366. Springer, Heidelberg (2005)
2. Tatusov, R.L., Fedorova, N.D., Jackson, J.D., Jacobs, A.R., Kiryutin, B., Koonin, E.V., Krylov, D., Mazumder, R., Mekhedov, S.L., Nikolskaya, A.N., Rao, B.S., Smirnov, S., Sverdlov, A.V., Vasudevan, S., Wolf, Y.I., Yin, J.J., Natale, D.A.: The COG database: an updated version includes eukaryotes. BMC Bioinformatics 4(1), 41 (2003)

3. Li, L., Stoeckert Jr., C., Roos, D.S.: OrthoMCL. Identification of Ortholog Groups for Eukaryotic Genomes. Genome Res. 13, 2178–2189 (2003)
4. Lee, Y., Sultana, R., Pertea, G., Cho, J., Karamycheva, S., Tsia, J., Parvizi, B., Cheung, F., Tonescu, V., White, J., Holt, I., Liang, F., Quackenbush, J.: Cross-referencing eukaryotic genomes: TIGR orthologous gene alignments (TOGA). Genome Research 12(3), 493–502 (2002)
5. Felsenstein, J.: PHYLIP - Phylogeny Inference Package (Version 3.2). Cladistics 5, 164–166 (1989)
6. Swofford, D.: LPAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4. Sinauer Associates, Sunderland, Massachusetts (2003)
7. Alexandros, S.: RAxML-VI-HPC: Maximum Likelihood-based Phylogenetic Analyses with Thousands of Taxa and Mixed Models. Bioinformatics 22(21), 2688–2690 (2006)
8. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. 15, 403–410 (1990)
9. Pruitt, K.D., Katz, K., Sicotte, H., Maglott, D.R.: Introducing RefSeq and LocusLink: curated human genome resources at the NCBI. Trends Genet. 16(1), 44–47 (2000)
10. Stajich, J.E., Block, D., Boulez, K., Brenner, S.E., Chervitz, S.A., Dagdigian, C., Fuellen, G., Gilbert, J.G.R., Korf, I., Lapp, H., Lehvaslaiho, H., Matsalla, C., Mungall, C.J., Osborne, B.I., Pocock, M.R., Schattner, P., Senger, M., Stein, L.D., Stupka, E.D., Wilkinson, M., Birney, E.: The Bioperl Toolkit: Perl modules for the life sciences. Genome Research 12(10), 1611–1618 (2002)
11. PBS Pro, `http://www.pbspro.com/`
12. Thompson, J.D., Higgins, D.G., Gibson, T.J.: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. Nucleic Acids Res. 22, 4673–4680 (1994)
13. Wang, J.T.L., Shan, H., Shasha, D., Piel, W.H.: TreeRank: A Similarity Measure for Nearest Neighbor Searching in Phylogenetic Databases. In: Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SSDBM 2003), Cambridge, Massachusetts, pp. 171–180 (2003)
14. Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming A POSIX Standard for Better Multiprocessing. O'Reilly, Sebastopol (1996)
15. Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 379–387. Springer, Heidelberg (2003)
16. Hall, C., Brachat, S., Dietrich, F.S.: Contribution of horizontal gene transfer to the evolution of Saccharomyces cerevisiae. Eukaryot Cell. 4(6), 1102–1115 (2005)

Efficient Parallelization of the Preconditioned Conjugate Gradient Method

Gilbert Accary[1], Oleg Bessonov[2], Dominique Fougère[3], Konstantin Gavrilov[4], Sofiane Meradji[3], and Dominique Morvan[5]

[1] Université Saint-Esprit de Kaslik, B.P. 446 Jounieh, Lebanon
[2] Institute for Problems in Mechanics of Russian Academy of Sciences, 101, Vernadsky ave., 119526 Moscow, Russia
[3] Laboratoire de Modélisation, Mécanique et Procédés Propres, L3M–IMT, La Jetée, Technopôle de Château-Gombert, 13451 Marseille Cedex 20, France
[4] Perm State University, 15, Bukirev str., 614990 Perm, Russia
[5] Université de la Méditerranée, UNIMECA, 60, rue Joliot Curie, 13453 Marseille Cedex 13, France

gilbertaccary@usek.edu.lb, bess@ipmnet.ru, fougere@l3m.univ-mrs.fr, gavrilov_k@inbox.ru, sofiane@l3m.univ-mrs.fr, dominique.morvan@univmed.fr

Abstract. In this paper we present methods for efficient parallelization of the solution of pressure Poisson equation arising in 3D CFD forest fire modeling. The solution procedure employs the Conjugate Gradient method with implicit Modified ILU (MILU) preconditioner. The basic idea for parallelizing recursive incomplete-decomposition algorithms is to use a direct nested twisted approach in combination with a staircase method. Parallelization of MILU-CG solver is implemented in OpenMP environment for Non-uniform memory (NuMA) computer systems. Performance results of the parallelized algorithm are presented and analyzed for different number of processors (up to 16).

1 Introduction

This work is performed in frame of the European integrated fire management project (Fire Paradox) and is aimed to simulate 3D fire behaviour and effects. The work is based on the previous experience in 2D simulations [1,2] extended to 3 spatial dimensions. This extension requires deep optimization of the solution procedure together with taking benefit of parallelization for moderate number of processors. At the initial stage of the work, this parallelization was implemented for the basic algorithm, resulting in fully functional code covering several physical mechanisms [3,4]. Efficient parallelization of the initial numerical code was possible because of the "explicit" nature of most numerical procedures including the Conjugate Gradient (CG) solver for pressure Poisson equation.

However, the original CG method [5,6] is expensive and consumes a lot of computing power (up to 50-70% of total processor time). The main reason of it is that CG has the same explicit nature and propagates information through the computational domain very slow, one grid point per iteration. In order to accelerate this method, a preconditioning procedure must be employed. Unfortunately, if such a procedure belongs to the explicit class, it can't accelerate the convergence substantially. Because of this, procedures of the implicit class must be introduces, such as the Incomplete LU-decomposition (ILU) method [6,7].

ILU-preconditioners have proved to be very efficient in solving symmetric positive-definite linear systems by CG. However, the property of fast propagation of information greatly complicates parallelization of these algorithms. Generally speaking, there is no universal and efficient method for parallelizing ILU, taking into account the sparse nature of a matrix. As a consequence, researchers have to develop some modifications of the method. One well-known approach is the class of domain decomposition methods [7], where the solution of the original global linear system for preconditioning is replaced with the independent solutions of smaller (local) systems within subdomains, with further coupling of partial results. Despite being very popular, this approach is not enough efficient because it makes the convergence slower (or even impossible at all).

Therefore, it is desirable to keep the basic idea of ILU and look for a direct method of parallelization of LU-decomposition. In order to achieve this goal, it is necessary to find all sorts of parallelization potential of the method. For some classes of numerical problems, this potential can be revealed from their geometric properties. For example, in case of one-dimensional discretization, the resulting tridiagonal linear system can be solved by the "twisted factorization" method which is easily parallelizable for 2 processors. For Cartesian grid in 3 dimensions, this method can be generalized with increasing the parallelization potential to 8 processors. The new method is known as "nested twisted factorization" [8].

In order to extend parallelization to 16 processors or more, we can use another approach, the "staircase method" [9] (also called "pipeline parallelization" [10]). Being not very efficient as a main parallelization method (see [11]), it can be used as a supplement to the basic approach (e.g. for doubling the number of processors used).

In the current work we utilize both above approaches: nested twisted factorization for parallelizing up to 8 processors, and, additionally, staircase extension for 16 processors.

Another point to analyze is the derivation of the Modified ILU (MILU) preconditioning [12]. The MILU preconditioner works much better than the original ILU (in terms of convergence rate) because it approximates the inverse of a matrix much more accurately. Usually, the MILU preconditioner is considered as a modification of the ILU by so-called "diagonal compensation", without looking at the quantitative effect of this compensation and without estimating approximation errors. In order to be able to derive a MILU-class preconditioner that is suitable for efficient parallelization it is necessary to carry out such analysis.

Thereby, in this paper we will describe the numerical method and general parallelization approach for NuMA computers in frame of OpenMP environment, then derive and analyze a MILU-class preconditioner, and present methods for parallelizing the MILU-CG solver for 8 and 16 processors. At the end, we will evaluate and discuss parallel performance of the new method.

2 Numerical Method and Main Parallelization Approach

We consider Newtonian fluid flow governed by non-stationary Navier-Stokes equations in Boussinesq and Low Mach approximations. The set of equations consists of the continuity equation, the momentum equations in three spatial dimensions ($i = 1, 2, 3$) and the equations for energy and turbulent quantities:

$$\frac{\partial}{\partial t}(\rho\phi) + \frac{\partial}{\partial x_i}(\rho\phi u_i) = \frac{\partial}{\partial x_i}\left(\mathit{\Gamma}\left(\frac{\partial\phi}{\partial x_i}\right)\right) + S_\phi \quad \text{with} \quad \phi = 1, u_1, u_2, u_3, T, k, \epsilon$$

where ϕ represents the transported variable; ρ and u_i are respectively the local density and the i-th component of velocity; $\mathit{\Gamma}$ – the effective diffusion coefficient; S_ϕ – the source term for the corresponding variable.

The Finite Volume discretization is applied to the non-uniform Cartesian staggered grid. The transport equations are solved by a fully implicit segregated method based on the SIMPLE-class algorithm. The non-symmetric linear systems obtained from the discretized equations are solved by the BiCGStab iterative method, while the symmetric linear system of the pressure Poisson equation is solved by the Conjugate Gradient method (CG). The code is applicable for simulation of flows in rectangular domains.

Parallelization of the algorithm is performed in frame of the OpenMP environment [13]. Generally, the OpenMP extension to a high level language (Fortran) is very simple and complements this language by several comment-like directives which instruct a compiler how to perform parallelization. The most important directive is "PARALLEL DO" which is usually applied to an outermost "`do`" statement (Fig. 1, left). In accordance with the number of processors, iterations of this loop are evenly distributed between threads of a program for execution in different processors. This corresponds to the geometric splitting of a processed array into sub-arrays by the last spatial dimension (Fig. 1, right).

The OpenMP parallelization model is very convenient for "true" shared-memory computers with uniform memory, because in this case it is possible to split a multidimensional computational domain by any spatial direction. For

```
!$OMP DO
      do K=1,Nz
        do J=1,Ny
          do I=1,Nx
      Wo3(I,J,K)=Wo2(I,J,K)+
     &      beta*Wo3(I,J,K)
          enddo
        enddo
      enddo
!$OMP END DO
```

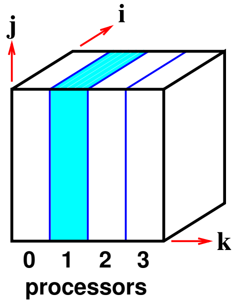


Fig. 1. Example of "PARALLEL DO" directive (left); geometric splitting of a data array by this directive (right)

systems with Non-uniform memory (NuMA), only splitting by the last direction ensures that necessary portions of data are fully located within the corresponding processor node's memory. In order to avoid remote memory accesses, algorithms must be rearranged. Some sorts of algorithms (e.g. "implicit", with recursive dependences in all spatial directions) can't be parallelized easily and efficiently within the OpenMP model. On the other hand, "explicit" algorithms that pass sequentially through data arrays and use small local data access patterns (stencils), may benefit from this model. Accesses to remote memory occur only within boundaries between subdomains in this case.

One-dimensional splitting of multidimensional arrays imposes another limitation on the OpenMP model for NuMA computers: subdomains may become too "narrow" in this dimension, and, as a result, accesses to remote memory through boundaries become frequent enough. Also, the last dimension may become not divisible by the number of processors that results in a bad load balance. These limitations restrict the degree of efficient parallelization by moderate number of processors (typically 8–16).

Parallelization method of the basic algorithms is described in details in [3].

3 Analysis of Implicit Preconditioners

The original (non-preconditioned) Conjugate Gradient method [5,6] 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 and requires about $O(N)$ iterations, where N is the dimension of the problem in one spatial direction.

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). To be more accurate, a system to be solved looks as $(L^{-1}AL^{-\mathrm{T}})\boldsymbol{x}^* = L^{-1}\boldsymbol{b}$ where $LL^{-1} = M$, but in the preconditioned CG algorithm only computations of the sort $\boldsymbol{x} = M^{-1}\boldsymbol{z}$ or $M\boldsymbol{x} = \boldsymbol{z}$ are needed [5,6,14].

Preconditioning works well if the condition number of the matrix $L^{-1}AL^{-\mathrm{T}}$ 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 (i.e. $\boldsymbol{x} = B\boldsymbol{z}$ is to be computed).

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 degenerates to the diagonal preconditioner $B = P^{-1}$ which is normally not considered as a true preconditioner because of its simplicity. For $n = 1$, the Jacobi preconditioner looks as $B = (I + (I - P^{-1}A))P^{-1}$ and improves acceleration rate by two times (with some increase of computational complexity). This exactly corresponds to the expansion of the computational stencil of one iteration of the algorithm. Therefore, it can be easily applied and parallelized. On the other hand, variants of Jacobi preconditioner with $n = 2$ or $n = 3$ happen to be not effective (acceleration is improved by less than 3 or 4 times, respectively) and are therefore rejected.

Unfortunately, neither sort of the simple explicit preconditioner can improve the convergence radically. For this reason, it is absolutely necessary to design a preconditioner of implicit sort.

Looking at the Incomplete LU (ILU) class of preconditioners, let us first consider basic criteria for its selection:

– Preconditioner matrix M must be chosen "close" to the main matrix A in such a way that the approximation error $\varepsilon = ||M\boldsymbol{x} - A\boldsymbol{x}||$ would be sufficiently small (for typical values of the solution vector $\boldsymbol{x}$).
– Matrix M must be suitable for decomposition into factors (e.g. $M = LU$) and these factors must be invertable 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 auxiliary systems must be subject to efficient parallelization (and decomposition of the matrix M also, if possible).

Let us now reformulate the concept of "incomplete (approximate) decomposition of the original matrix A" as "exact decomposition of the approximating matrix M". With this new formulation, we will need to find such factors L and U (where $M = LU$) that difference between matrices A and M is minimized, provided that these factors are easy to invert.

The most popular decomposition for ILU is splitting an approximation of the symmetric sparse matrix A by lower triangular factor L and upper triangular factor $U = L^{\mathrm{T}}$. In this case, the product $M = LL^{\mathrm{T}}$ will reproduce mainly the sparsity pattern of A, generating additionally some fill-in. For 7-diagonal Poisson matrix in 3D we will have 6 parasitic diagonals. Influence of these parasitic diagonals must be compensated by some way.

Further consideration will be carried out for 2-dimensional case (extension to 3D is straightforward). Multiplying together 3-diagonal triangular factors L and L^{T}, we obtain a 7-diagonal matrix M with 2 parasitic diagonals (Fig. 2). If we depict stencils of all three matrices, we will see that the stencil of the product matrix M has acquired two new nodes `SE` and `NW` corresponding to the lower and upper parasitic diagonals. In terms of the computational grid, these nodes correspond to grid points `(i+1,j-1)` and `(i-1,j+1)`, respectively (where the polar node `P` corresponds to `(i,j)`).

If the solution vector $\boldsymbol{x}$ is sufficiently smooth, we can approximate values in the nodes `SE` and `NW` by different ways (estimating the approximation error in each case).

1. Neglect: $x_{\mathtt{SE}} = 0$, $x_{\mathtt{NW}} = 0$ $(\varepsilon = ||M\boldsymbol{x} - A\boldsymbol{x}|| = O(h^0))$. This is the original unmodified ILU (DILU). Its convergence properties are not good, they are at the same level as that of the explicit preconditioners ($O(N)$ iterations are required). For this reason, DILU normally is not used for finding smooth solutions.

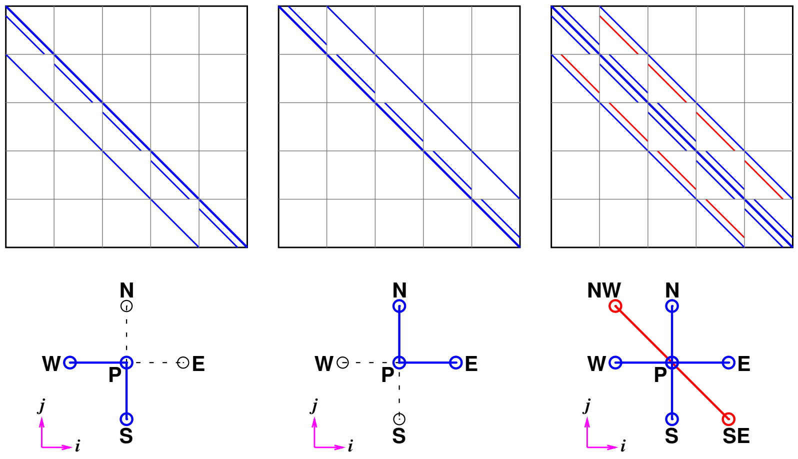


Fig. 2. Illustration of the decomposition $L{\cdot}L^{\mathrm{T}} \rightarrow M$ (above) and corresponding stencils of this decomposition (below)

2. Piecewise constant approximation: $x_{\mathtt{SE}} = x_{\mathtt{P}}$, $x_{\mathtt{NW}} = x_{\mathtt{P}}$ ($\varepsilon = ||M\boldsymbol{x} - A\boldsymbol{x}|| = O(h^1)$). This is exactly the Modified ILU (MILU) we are looking for. Its convergence properties are much better ($O(N^{\frac{1}{2}})$ iterations are required). For this reason, MILU is widely used.

3. Bilinear interpolation: $x_{\mathtt{SE}} = x_{\mathtt{S}} + x_{\mathtt{E}} - x_{\mathtt{P}}$, $x_{\mathtt{NW}} = x_{\mathtt{N}} + x_{\mathtt{W}} - x_{\mathtt{P}}$ ($\varepsilon = ||M\boldsymbol{x} - A\boldsymbol{x}|| = O(h^2)$). This is similar to the Strongly Implicit Procedure (SIP) of Stone [15]. This procedure is generally used as a principal iterative method, rather than as a preconditioner for CG. Unfortunately, this method can't be applied as a preconditioner to the classical (symmetric) Conjugate Gradient because it produces non-symmetric matrix M. However, it can be considered as an option for non-symmetric solvers of CG family.

Thus we have obtained a quantitative foundation of the Modified ILU method. Usually, the MILU is considered in terms of diagonal compensation [12,6]. The above formulas for $x_{\mathtt{SE}}$ and $x_{\mathtt{NW}}$ justify this approach, because approximations of these values as $x_{\mathtt{P}}$ exactly correspond to the application of this compensation to the main diagonal. It should be noted that in complicated cases it is necessary to be accurate and attentive when applying this compensation in order to keep the order of approximation of the matrix M. It seems that researchers sometimes don't achieve the expected convergence rate with MILU and conclude that it is not much better for their problems than the original unmodified ILU.

To perform MILU decomposition and compute its triangular factor L, we apply the following formula (where brackets indicate the above approximation):

$$LL^{\mathrm{T}}\boldsymbol{x} \approx \left[LL^{\mathrm{T}}\boldsymbol{x}\right]_{\mathrm{approx}} = A\boldsymbol{x}$$

Practically, it is convenient to represent the decomposition as follows:

$$M = (L + D)D^{-1}(D + L^{\mathrm{T}})$$

where D – main diagonal, and L and L^{T} – non-diagonal elements of triangular factors. With such decomposition, these non-diagonal elements will be equal to the non-diagonal elements of the original matrix A (L_A and L_A^{T}, respectively), and it will be necessary to compute only diagonal elements D.

Generally, L and L^{T} in the above form of decomposition don't have to be triangular matrices. In order to approximate the matrix A, it is enough if the following conditions are satisfied:

$$\begin{aligned} L \cup L^{\mathrm{T}} &= L_A \cup L_A^{\mathrm{T}} \\ L \cap L^{\mathrm{T}} &= \emptyset \end{aligned}$$

In other words, these factors must complement each other in representing non-diagonal elements of A. For this general form of decomposition, parasitic elements of the matrix M, as well as new nodes in the matrix stencil, may appear in different positions.

The general form of incomplete decomposition (that can't be called "Lower-Upper" anymore) may become convenient for constructing matrices of special form suitable for efficient parallelization. For example, the nested twisted decomposition (that will be considered in the next section) produces factors and a product matrix as on Fig. 3 (here represented for 2D case).

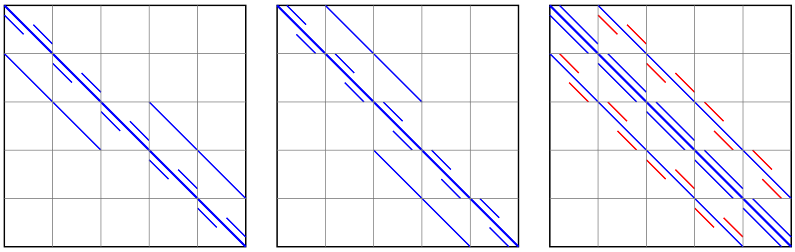


Fig. 3. Nested twisted factorization $L \cdot L^{\mathrm{T}} \rightarrow M$ suitable for parallelization

Orientations of new nodes appeared in stencils of product matrices for the classical LU-decomposition, and for the above nested twisted form, are depicted on Fig. 4, left. Here, stencils are shown in four quadrants of 2D computational domain for both cases, at left and at right respectively (all stencils within a quadrant have the same orientation). We can see that for the classical decomposition (Fig. 2) all stencils within a domain have the same orientation, while for the nested twisted form (Fig. 3) there is a symmetry in both (all) spatial directions. This symmetry may happen to be useful for improving convergence in some cases. Fig. 4, right, represents convergence history of two methods – original LU (anisotropic) and nested twisted (symmetric) – for non-smooth grid $100 \times 100 \times 100$ with strong non-uniformities. Here the symmetric method has convergence rate about 1.5 times higher than the non-symmetric one.

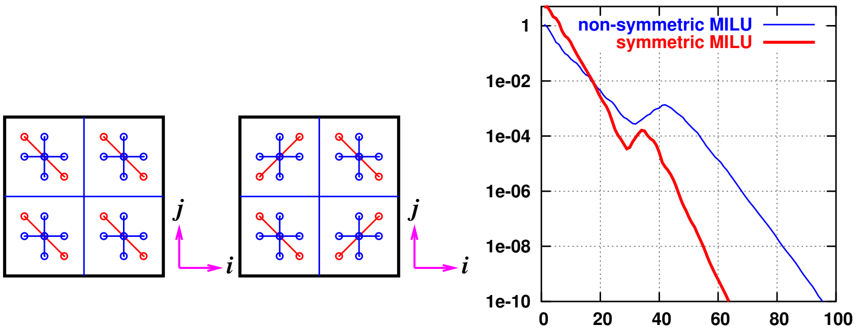


Fig. 4. Orientation of stencils of classical (non-symmetric) and nested twisted (symmetric) factorization (left); example of convergence of these two factorizations (right)

Table 1 presents the comparison of 3 discussed preconditioners on the solution of discretized Poisson equation of the size $200 \times 200 \times 120$ ($4.8 \cdot 10^6$ grid points) with accuracy 10^{-10} on Itanium 2 processor (1.5 GHz, L3-cache 4M). Presented results confirm acceleration factors in comparison with classical (diagonal-preconditioning) case: 2 for Jacobi, and about $N^{\frac{1}{2}}$ for Modified ILU.

Table 1. Convergence and computational complexity of preconditioners

preconditioner	classical diagonal	polynomial Jacobi n=1	MILU with symmetry
iteration count	346	172	37
processor time (sec)	58.55	38.55	10.81
relative speed	1.00	1.52	5.42
iteration cost (sec)	0.169	0.224	0.292
relative iteration cost	1.00	1.32	1.73

4 Parallelization Method for 8 Processors

Decomposition of the approximating matrix $M = (L + D)D^{-1}(D + L^{\mathrm{T}})$ as well as solution of auxiliary linear systems $(L + D)D^{-1}\boldsymbol{y} = \boldsymbol{z}$ and $(D + L^{\mathrm{T}})\boldsymbol{x} = \boldsymbol{y}$ belong to the class of implicit (recursive) algorithms. For this reason a geometric domain can't be split into subdomains to perform computations independently in different processors. Therefore, it is necessary to find such geometric properties of the algorithm that parallelization would become possible.

The original idea is taken from the twisted factorization of a tridiagonal linear system, when Gauss elimination is performed from two sides simultaneously (for a subdiagonal and a superdiagonal, respectively). This idea can be naturally generalized to 2 and 3 dimensions, as follows from the fact that all spatial directions are symmetric to each other. Owing to this, nested application of twisted factorization becomes possible. This method is called "nested twisted" [8] and is sometimes formulated as "van der Vorst ordering of a matrix" (see [14]).

The nested twisted factorization method can be used for direct parallelization of the solution for up to 8 processors (in a Cartesian domain). The computational scheme of this method is as following. A rectangular parallelepipedic domain is split into 8 octants by separator planes (Fig. 5). In each octant, Gauss elimination is performed from the corner in the direction inwards (in all 3 dimensions), independently in different processors (Fig. 5, left). This sort of Gauss elimination, as well as the initial decomposition of a matrix (performed once in the beginning of the routine) corresponds to the matrix splitting depicted on Fig. 3 (as represented for 2D case).

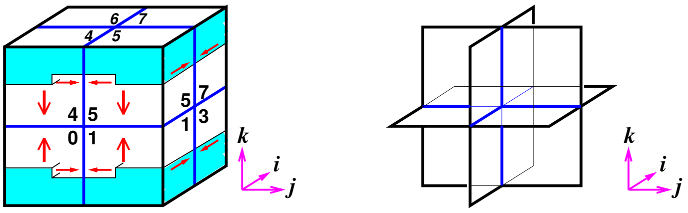


Fig. 5. Parallelization of the nested twisted factorization. Illustration of the method (left); separator planes (right).

After finishing eliminations in the internal points of octants, they are performed in quadrants of separator planes by the same way (Fig. 5, right). Then, points on lines of intersection of separator planes are processed, and finally a solution at the central point is computed. The following backsubstitution is performed in reverse sequence, from the central point in the direction outwards.

The above scheme needs some reorganization of data arrays when implemented on computer systems with non-uniform memory (NuMA). The natural geometric splitting of arrays by the last spatial direction (Fig. 6, right) is no more applicable. It would lead to multiple accesses to remote memories, because such array placement doesn't correspond to the data access pattern of the algorithm (Fig. 6, left). Accesses to data located within a local memory of another processor are performed with larger latency and lower data access rate, that would lead to significant performance degradation.

Besides this, accesses must be organized monotonically, by reading long sequences of data with increasing or decreasing addresses. This means that (at least) first two indices of arrays `(i,j)` both should be either increased or decreased through iterations of the main loop. The reason of this requirement is that modern processors rely on so-called streaming data prefetch, otherwise data access rate would be very low.

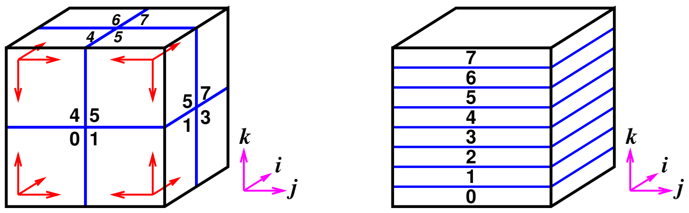


Fig. 6. Parallelization of the nested twisted factorization. Organization of data arrays (left) in comparison with the natural splitting (right).

As a consequence, all data arrays in the parallelized algorithm must be organized in such a way, that elements within an octant are numbered from the corner in the direction inwards. Technically, these data are represented as 4-dimensional arrays `(i,j,k,ij)` where the first 3 indices enumerate elements within a subarray (octant), while the last one indicates the subarray (octant) number (Fig. 6, left). This placement requires that initial data arrays (that are in the natural ordering) are copied to the work arrays with the above organization in the beginning of the routine, and resulting data are copied back in the end.

5 Extension of Parallelization Method for 16 Processors

Parallelization for 16 processors or more needs another approach because the potential of nested twisted factorization is exhausted. For recursive algorithms as Gauss elimination, the staircase (or pipeline) method can be employed [9,10]. This method is illustrated on Fig. 7. The method is applied within each octant in order to additionally parallelize processing for 2 (or more) processors. For the efficient implementation, a subdomain is split into 2 parts in the direction of the index `j` – see, for example, the bottom-left octant (Fig. 7) divided between processors 0 and 1. Computations in a plane `(i,j)` for any particular value of `k` can't be performed by processor 1 until they've been finished by processor 0. However they can be fulfilled in a pipelined fashion: processor 1 computes a layer for some `k` at the same time when processor 0 computes the next layer for `k+1`. This method needs synchronization between processors in a pair: before starting computations for some `k`, processor 1 must wait for processor 0 to finish computations in the same layer. At the backsubstitution stage of the algorithm, computations are performed in reverse order, when processor 0 waits for processor 1 for synchronization.

Implementation of this method leads to some algorithmic overhead because at the beginning (for the first value of `k`) processor 1 is idle waiting for the results from processor 0, and at the end (for the last value of `k`) processor 0 is idle after finishing its work. In order to reduce synchronization expenses, blocking by the index `k` can be applied. Currently, the blocking factor it equal to 2.

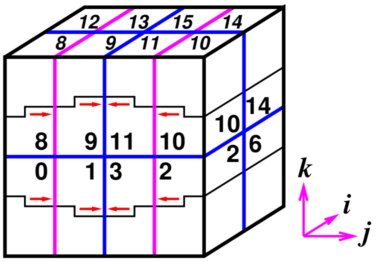


Fig. 7. Parallelization for 16 processors with the staircase method

The staircase method must be used only in the recursive parts of the algorithm, i.e. in the application of incomplete decomposition. The Conjugate Gradient algorithm itself is not recursive and doesn't need synchronizations between processors in pairs.

The above method of parallelization can be used for more than two processors. In this case the algorithmic scheme will have more "stairs" and more points of synchronizations. However, the staircase approach is not widely used alone as a main parallelization method because of performance limitations (see [11]).

6 Parallelization Results

The results of parallelization efficiency of the new method are presented on Fig. 8. These results correspond to the solution of discretized Poisson equation of the size $200 \times 200 \times 120$ ($4.8 \cdot 10^6$ grid points) with accuracy 10^{-10} on SGI Altix 350 computer system with non-uniform memory organization (NuMA).

	No. of processors				
	1	2	4	8	16
time (seconds)	10.81	8.54	4.47	2.43	1.47
relative speedup	–	1.25	1.93	1.84	1.66
total speedup	–	1.25	2.42	4.45	7.35

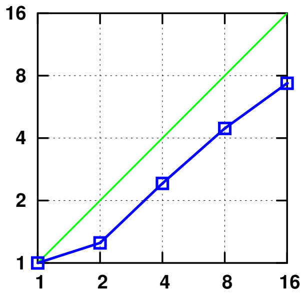


Fig. 8. Parallelization results

Before commenting these results, we will first describe a computer system used for development and measurements. SGI Altix 350 system is organized of two-processor computational nodes interconnected by the special NuMA-link interfaces through the high-speed switch that provides accesses to non-local (remote) memories. Logically, the considered system belongs to the shared-memory class, when every process may transparently access any memory location in a system. However, remote accesses are much slower than local ones.

Another point is that two processors within a node share the same memory and compete for accesses. As a result, total memory access rate achieved by two processors is not much higher than the rate achieved by one, i.e. common memory is the main performance limiter for memory-bound applications. The ILU-preconditioned CG algorithm belongs to this class, and for this reason the achieved acceleration for 2 processors is rather small, only 1.25. However, if 2-processor run is performed on 2 computational nodes, using one processor in each node, its acceleration becomes almost optimal and reaches 1.93. This means, that low results for 2 processors represent the property of a computer system, rather than that of the parallelization method. On computer systems with higher memory access limit, it is expected that results would be substantially higher.

For 4 processors, the achieved acceleration (relative to 2 processors) is good enough, because each pair of processors works with its individual memory and total memory throughput doubles. For 8 processors, we also have the reasonable scaling, while for 16 processors we can see negative effects of algorithmic overhead and non-perfect load balance.

Generally, despite the increased complexity of parallelization of the implicit incomplete decomposition, parallelization results of the current implementation are in good correspondence with the results for the basic algorithm [3] which is of the explicit nature. Owing to the substantial increase of the speed of CG method (due to the MILU-class preconditioning), as well as to the efficient parallelization, the relative cost of the solution of Poisson equation within the total processor time of the problem has decreased sharply and dropped to the insignificant level.

7 Conclusion

In this work we have developed a parallel method of the solution of pressure Poisson equation based on the generalized idea of Modified ILU preconditioning for the Conjugate Gradient procedure. The new method demonstrates good parallelization efficiency on multiprocessor systems with non-uniform memory organization (NuMA) with up to 16 processors. This sort of systems becomes more and more popular with development and propagation of multi-core microprocessors. The new generation of quad-core processors, Intel Xeon (Core i7) and AMD Opteron (K10), will become the main computational device in coming years, with integration of 2- and 4-processors in a single system of the type NuMA (up to 16-32 processor in total). This sort of computer systems will have more balanced memory subsystems with much higher data access bound, allowing the above method to achieve better parallel performance and scalability. The described method can be also implemented in the hybrid MPI/OpenMP environment for modern clusters built on computing nodes with NuMA organization.

Acknowledgements. This work was supported by the European integrated fire management project (Fire Paradox) under the Sixth Framework Programme (Work Package WP2.2 "3D-modelling of fire behaviour and effects"), and partly by the Programme N 15 of the Presidium of the Russian Academy of Sciences.

References

1. Morvan, D., Dupuy, J.L.: Modeling of Fire Spread through a Forest Fuel Bed Using a Multiphase Formulation. Combust. Flame 127, 1981–1994 (2001)
2. Morvan, D., Dupuy, J.L.: Modeling the Propagation of a Wildfire through a Mediterranean Shrub Using a Multiphase Formulation. Combust. Flame 138, 199–210 (2004)
3. Accary, G., Bessonov, O., Fougère, D., Meradji, S., Morvan, D.: Optimized Parallel Approach for 3D Modelling of Forest Fire Behaviour. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 96–102. Springer, Heidelberg (2007)
4. Scarella, G., Accary, G., Meradji, S., Morvan, D., Bessonov, O.: Three-Dimensional Numerical Simulation of the Interaction between Natural Convection and Radiation in a Differentially Heated Cavity in the Low Mach Number Approximation. In: CHT 2008 International Symposium on Advances in Computational Heat Transfer, CHT-08-193, Begell House, Inc. (2008)
5. Shewchuk, J.R.: An Introduction to the Conjugate Gradient Method without the Agonizing Pain. School of Computer Science, Carnegie Mellon University, Pittsburgh (1994)
6. Ortega, J.M.: Introduction to Parallel and Vector Solution of Linear Systems. Plenum Press, New York (1988)
7. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS Publishing, Boston (2000)
8. van der Vorst, H.A.: Large Tridiagonal and Block Tridiagonal Linear Systems on Vector and Parallel Computers. Par. Comp. 5, 45–54 (1987)
9. Bastian, P., Horton, G.: Parallelization of Robust Multi-Grid Methods: ILU-Factorization and Frequency Decomposition Method. SIAM J. Stat. Comput. 12, 1457–1470 (1991)
10. Elizarova, T., Chetverushkin, B.: Implementation of Multiprocessor Transputer System for Computer Simulation of Computational Physics Problems (in Russian). Mathematical Modeling 4(11), 75–100 (1992)
11. Vuik, C., van Nooyen, R.R.P., Wesseling, P.: Parallelism in ILU-Preconditioned GMRES. Par. Comp. 24, 1927–1946 (1998)
12. Gustafsson, I.: A Class of First Order Factorization Methods. BIT 18, 142–156 (1978)
13. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering 5(1), 46–55 (1998)
14. Benzi, M.: Preconditioning Techniques for Large Linear Systems: A Survey. J. Comput. Phys. 182, 418–477 (2002)
15. Stone, H.L.: Iterative Solution of Implicit Approximations of Multidimensional Partial Differential Equations. SIAM J. Numer. Anal. 5, 530–558 (1968)

Parallel FFT with Eden Skeletons[⋆]

Jost Berthold, Mischa Dieterle, Oleg Lobachev, and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{berthold,dieterle,lobachev,loogen}@informatik.uni-marburg.de

Abstract. The paper investigates and compares skeleton-based Eden implementations of different FFT-algorithms on workstation clusters with distributed memory. Our experiments show that the basic divide-and-conquer versions suffer from an inherent input distribution and result collection problem. Advanced approaches like calculating FFT using a parallel map-and-transpose skeleton provide more flexibility to overcome these problems. Assuming a distributed access to input data and re-organising computation to return results in a distributed way improves the parallel runtime behaviour.

1 Introduction

The well-known Fourier transform, which describes frequency distribution in a signal, finds diverse applications from pure mathematical applications to real-life scenarios such as digital signal processing. Today's state of the art is the *Fast Fourier Transform* (FFT). Cooley and Tukey [4] were the first to propose an FFT algorithm in 1965 (known as 2-radix FFT) with time complexity $O(n \log n)$. A range of other FFT algorithms have been discovered since then [16].

Developing an efficient parallel distributed-memory implementation of FFT is a great challenge. The manual of the recent 3.2 alpha release of FFTW[1] warns that "distributed-memory parallelism can easily pose an unacceptably high communications overhead for small problems". In the broader context of an implementation for parallel computer algebra algorithms in the parallel Haskell extension Eden [15,13], we investigate parallelisation strategies for different FFT algorithms. The goal of our work has not been to develop the fastest distributed-memory FFT, but to investigate a skeleton-based parallelisation of FFT. In Eden, skeletons [3,14,1] are higher-order functions defining general parallel evaluation schemes. The skeleton approach to parallelisation cleanly separates problem-related and problem-independent issues. This simplifies the parallelisation of algorithms enormously. In essence, FFT algorithms are based on divide-and-conquer strategies. In this paper, we utilize skeletons for two variations of parallel divide-and-conquer evaluations: a *distributed expansion* scheme which unfolds the computation tree dynamically and spawns

⋆ Supported by the DFG grant LO 630-3/1.

[1] Fastest Fourier Transform in the West, `http://www.fftw.org/fftw-3.2alpha3-doc/`

parallel processes for the evaluation of sub-trees as long as processor elements are available, and a *flat expansion* scheme which unfolds the tree up to a given depth and evaluates all sub-trees at this depth in parallel. Moreover, we present a parallel *map-and-transpose skeleton* for the implementation of more advanced FFT methods. Our skeletons are applicable to a whole *class* of algorithms, those which rely on fixed-branching divide-and-conquer or parallel map-and-transpose schemes.

We analyse the parallel runtime behaviour of various skeleton/algorithm combinations using activity profiles of parallel program executions on networks of workstations, i. e. distributed-memory parallel machines. In addition, we investigate their scalability when increasing the number of processor elements.

Plan of Paper. The following two sections elaborate on divide-and-conquer approaches of parallel FFT (Section 2) and on advanced approaches (Section 3). In each section, we will describe appropriate skeletons for the parallelisation of FFT algorithms and an experimental evaluation of the parallelised algorithms. Section 4 discusses related work, the final section concludes.

2 Divide-and-Conquer FFT

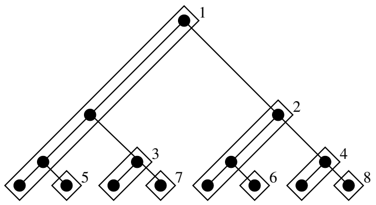
FFT Algorithms. The classic *2-radix FFT algorithm by Cooley and Tukey* divides the input vector xs of length n into two halves, computes their element-wise sum and difference, and multiplies the latter with powers of an n-th primitive root of unity, the *twiddle factors*. The algorithm recursively computes the FFT of these vectors, and combines the results simply by interleaving them element-wise. Recursion ends at singleton vectors which are returned unmodified. This version is called *decimation in frequency*.

An alternative version, called *decimation in time*, essentially consists of the opposite dividing and combining steps. The input vector is split into the sub-vectors with even and odd indices (inverse to the interleaving step above). After evaluating the recursive calls of FFT for the sub-vectors, the more complex combination of the result lists follows. The first and second half of the overall result are defined as element-wise sums and differences including again a multiplication with the twiddle factors.

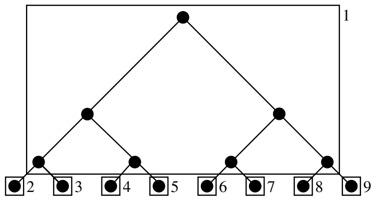
Divide-and-Conquer Skeletons. The essence of a divide-and-conquer algorithm is to decide whether the problem is trivial and, in this case, to solve it, or else to decompose non-trivial problems into a number of sub-problems, which are solved recursively, and to combine the output. A general skeleton takes parameter functions for this functionality, as shown here:

```
type DivideConquer a b = (a -> Bool) -> (a -> b)      -- trivial? / solve
                      -> (a -> [a]) -> ([b] -> b) -- split / combine
                      -> a -> b                   -- problem / result
```

The resulting structure is a tree of task nodes where child nodes are the sub-problems, the leaves representing trivial tasks.



(a) Binary distributed expansion, depth 3



(b) Binary flat expansion, depth 3

Fig. 1. Divide-and-conquer expansion schemes

A fundamental Eden skeleton which specifies a general divide-and-conquer algorithm structure can be found in [14]. In [1], we have refined and adapted this skeleton for fixed branching divide-and-conquer algorithms like FFT. Two different basic strategies have been used to unfold a process tree. The *distributed expansion scheme* creates the process tree in a *distributed* fashion: One of the tree branches is processed locally, the others are instantiated as new processes, as long as processor elements are available. This results in a *distributed expansion* of the computation (cf. Fig. 1(a)). Explicit placement of processes is essential to achieve a balanced distribution of processes on the available processor elements. The boxes indicate which tree node are evaluated by the same process. The numbers indicate a possible placement on 8 processor elements (PEs). The corresponding skeleton has the following interface (type):

```
dcN :: (Trans a, Trans b) =>
       Int -> [Int] ->    -- branching degree / processor elements
       DivideConquer a b
```

The Eden type class `Trans` provides internally used communication functions. The first two skeleton parameters determine the fixed branching degree of the underlying divide-and-conquer tree and a list of available processor numbers used for explicit process placement.

In the *flat expansion skeleton*, the main process unfolds the divide-and-conquer tree up to a given depth, usually with more branches than available PEs. The resulting subtrees are then evaluated by parallel processes, the main process combines the results of the sub-processes. This results in a homogeneous *flat expansion* scheme from a single source depicted in Fig. 1(b) for the binary variant. A uniform distribution of the subtrees on processors can be achieved using a farm of worker processes with static or dynamic task distribution. The corresponding skeleton has the following interface (type):

```
dcDM_N :: (Trans a, Trans b) =>
          Int -> Int ->    -- unfolding depth / branching degree
          DivideConquer a b
```

Here the first two skeleton parameters determine the unfolding depth of the underlying divide-and-conquer tree and the fixed branching degree. A detailed

```
-- Parallel 2-radix FFT, decimation in time, with input
-- chunking size, instantiates dcN skeleton
fft2radixTime    :: Int -> [Complex Double] -> [Complex Double]
fft2radixTime c xs
  = chunkDC c chunkL concat
     (dcN 2 [2..noPe]) isSingleton id (unshuffle 2) combine2
```

Fig. 2. Parallelisation by Skeleton Instantiation

explanation of these Eden divide-and-conquer skeletons can be found in [1]. Fig. 2 shows a sample instantiation of the `dcN` skeleton with branching degree 2 and explicit process placement on PEs 2 to `noPe` (the number of available PEs). Input vectors (lists) are chunked into larger pieces to reduce communication costs.

Experimental Results. The following runtime experiments have been performed on a local network of 8 Linux workstations with Core 2 Duo processors and 2 GB RAM connected by Fast Ethernet. The Eden runtime system is instrumented in such a way that a runtime flag activates a tracing mechanism which protocols parallelism-related events like process/thread creation/termination, state changes of machines (i. e. processors), processes and threads, and message sending and receiving. The trace files can then be visualised by the EdenTV tool (Eden Trace Viewer) [2]. The resulting graphics (see e.g. Figure 3) which are best viewed in colour are two-dimensional diagrams. The time scale is on the horizontal axis. The vertical axis shows the machine numbers, on which the processes are placed. For each process, there is a coloured horizontal bar, which shows the process states over time. Green parts (grey) indicate that a thread is working, red parts (dark grey) indicate that all threads of the process are blocked, usually because they are waiting for input, or because the processor is communicating. Yellow areas (light grey) indicate that there are runnable threads but some system activity like e. g. garbage collection is taking place. Data transfer, i. e. messages can be optionally indicated as arrows from the sending to the receiving process.

Our first experiments tested the standard Cooley-Tukey 2-radix FFT algorithm variants decimation in frequency and decimation in time with the distributed expansion and flat expansion skeletons. Figure 3 shows typical traces and the runtimes obtained with the following parameters: input size 2^{20} (double precision complex numbers), chunk size[2] 1500, recursion depth 4 and heap size 1500MB.

The activity profiles in Figure 3 reveal that the flat expansion skeleton leads to a much better runtime behaviour than the distributed expansion skeleton. This is due to the good load balance in the worker processes which start immediately. Note that the skeleton even co-locates one worker process with the main

[2] The chunk size is only used by the distributed expansion skeleton to reduce the number of messages.

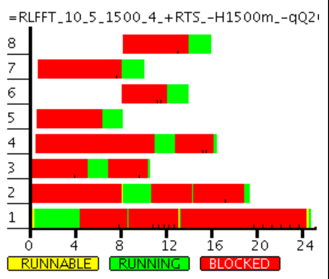
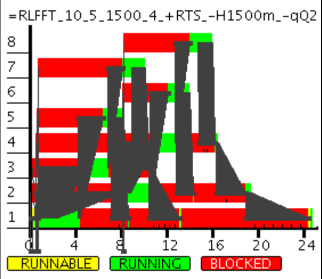
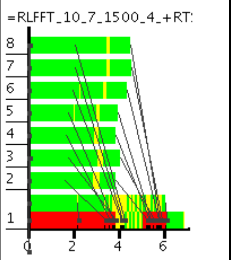
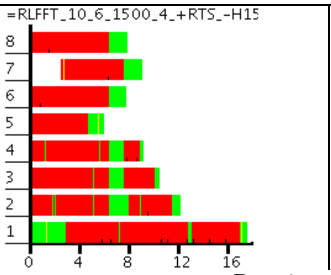
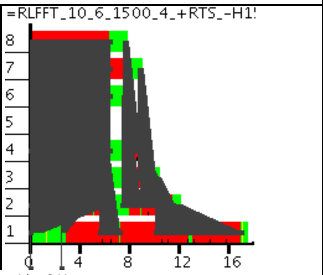
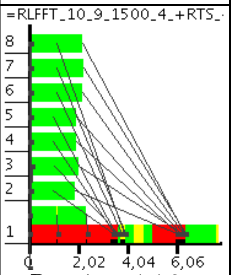
	Distributed Expansion		Flat Expansion
Decimation in Frequency			
	Runtime: 24.80 s		Runtime: 6.92 s
Decimation in Time			
	Runtime: 17.85 s		Runtime: 7.78 s

Fig. 3. Traces and runtimes of divide-and-conquer approaches, without/with messages

process on machine 1 (lowest bars). The communication overhead is low — only 80 messages were sent in both versions.

The decimation in frequency flat expansion version was the fastest version with 6.92 s. This is due to the fact that the post processing in the master can be done very fast, because combining the results is a trivial shuffle, while the top level combining phase of the decimation in time version takes almost three quarters of the overall runtime.

With the flat expansion skeleton, we eliminate the input communication, i. e. distributing tasks to the worker processes. Each worker receives the whole unevaluated task specification and evaluates its own part on demand. Contrarily, work distribution is slower with the distributed expansion skeleton because the main process distributes the tasks to all worker processes. These are initially blocked waiting for their tasks and start working at different points in time. This leads to an inhomogeneous runtime behaviour.

3 Advanced Approaches

The parallel divide-and-conquer FFT-implementations show an acceptable performance using few processors, but do not scale well. Therefore we have implemented a more sophisticated algorithm taken from [7] which minimizes data dependencies and provides more fine grained parallelism. The input vector is divided into rows of a matrix with side lengths $l = 2^k$. Thus, the input vector is of length $n = l^2 = 4^k$. The algorithm consists of three phases:

1. preprocessing: permutation of input in bit reverse order, tagging input elements with their position and their segment's length, split into rows.
2. central processing: local `fft3` $\circ$ a global transpose $\circ$ local `fft3`
3. postprocessing: concat and remove tags

The key difference between the ordinary sequential FFT and `fft3` is that the latter operates on triples which contain additional information like a position tag. It works with *global* twiddle factors to simulate a contiguous, single-dimensional FFT algorithm. The `divide` step is a trivial split of lists. The combine step needs to be modified using the additional information in the triples. Because of the permuted input, it is possible to perform FFT locally on the available subsets of global lists in a global manner. For more details, see [7].

We have derived a skeleton for the central phase of the above scheme which consists of a composition of parallel `maps` and an intermediate global communication to implement the global transpose. The skeleton has been inspired by the distributable homomorphism skeleton of Gorlatch and Bischof [8]. It can also be used for the distributed-memory FFT algorithms proposed in [17,10].

The Parallel Map-and-Transpose Skeleton implements the functionality

`(parMap f1)` $\circ$ `transpose` $\circ$ `(parMap f2).`

Defining it with this simple function composition is not appropriate, because all data would be gathered in the main process in between the two parmap phases. This again would provoke too much communication and process creation overhead. Our skeleton `parMapTranspose` includes a distributed transpose phase in between two parallel map evaluations. The skeleton's input is a matrix which will be distributed row cyclic. In our application the functions `f1` and `f2` will be sequential `fft3` invocations. In order to save the costs of input distribution, the parallel maps are executed by direct mapping [12] which means that the matrix is not communicated but transferred unevaluated within the process abstraction's body. The child processes will then evaluate the needed parts locally and demand driven.

The Eden code of the skeleton uses the following Eden constructs. The library function `spawn :: (Trans a, Trans b) => [Process a b] -> [a] -> [b]` creates a list of processes from a list of process abstractions and a list of corresponding process inputs. A process abstraction is a function that will be evaluated by a process. `Process` is the type constructor for Eden Process abstractions, which are created by the function `process :: (a -> b) -> Process a b`. Eden provides the following functions to dynamically define new input channels for processes. The Eden function `createChans :: Int` → `([ChanName a], [a])` creates a list of new (input) channel names. Data (lazily) received via the channels can be accessed in the second component of the result tuple of `createChans`. Channel names can be communicated to other processes which can write into the corresponding channels with the Eden function `multifill :: [ChanName a]` → `[a]` → `b` → `b`, which concurrently passes data via given channels and returns its third argument.

```
parMapTranspose :: Int -> ([a] -> [b]) -> ([b] -> [c]) -> [[a]] -> [c]
parMapTranspose np f1 f2 matrix = shuffle res
  where
   myProcs css = spawn [ process (distr2d_f np f1 f2 rows)
                       | rows <- unshuffle np matrix ] css
   (res,chanss) = myProcs $ transpose chanss

distr2d_fs ::  Int -> ([a] -> [b]) -> ([b] -> [c]) ->
              [[a]] -> [ChanName[b]] -> ([[c]],[ChanName [b]])
distr2d_fs np f1 f2 rows theirChanNs
  = let (myChanNs, theirFstRes) = createChans np
        intermediateRes  = map f1 rows
        myFstRes         = unshuffle np $ transpose intermediateRes
        res              = map f2 $ shuffleMatrixFracs theirFstRes
    in  (multifill theirChanNs myFstRes $ res, myChanNs)

-- types of auxiliary functions
-- round robin distribution and combination of list elements
unshuffle :: Int -> [a] -> [[a]]
shuffle   :: [[a]] -> [a]
-- combine n matrix fragments into one matrix
shuffleMatrixFracs :: [[[a]]] -> [[a]]
```

Fig. 4. Parallel map-and-transpose skeleton

The code of the parallel map-and-transpose skeleton `parMapTranspose` is shown in Figure 4. The distributed `map` functionality is easily defined. Let `np` be the number of available PEs (processing elements). We divide the matrix rows into `np` contiguous blocks using the function `unshuffle`. At the end the final result is re-composed using the inverse function `shuffle`. As many processes as available PEs are created using the Eden function `spawn`. Each process applies the function `distr2d_fs np f1 f2` to its portion of rows and the lazily communicated input (a row of `css`). The latter consists of a list of `np` channel names which are used to establish a direct link to all processes: each process can thus send data directly to each other process[3]. Each process evaluates the function `distr2D_fs` which firstly leads to the creation of `np` input channel names `myChanNs` for the corresponding process. These are returned to the parent process in the second component of the result tuple of `distr2d_fs`. The parent process receives a whole matrix `chanss :: [[ChanName a]]` of channel names (`np` channel names from `np` processes), which it transposes before sending them row-wise back to the child processes. Each process receives thus lazily `np` channel names `theirChanNs` for communicating data to all processes. The parallel transposition can thus occur without sending data through the parent process.

[3] To simplify the specification the channel list even contains a channel which will be used by the process to transfer data to itself.

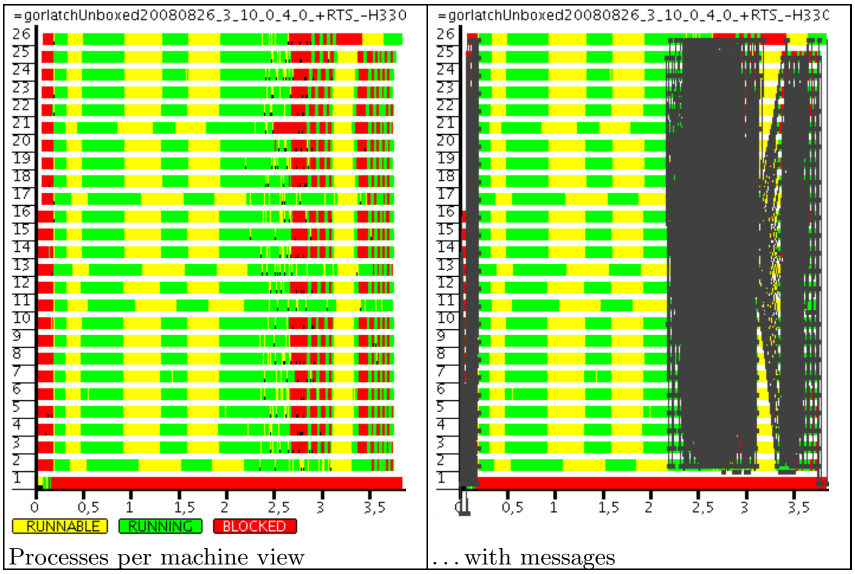


Fig. 5. Trace of parallel FFT using map-and-transpose skeleton (input size 2^{20}, 3.5 seconds on 26 Pentium 4 machines)

After the first `map f1` evaluation, a process locally `unshuffle`s the columns of the result (the locally transposed result rows) into `np` lists. These are sent via the received input channels of the other processes using the function `multifill`. The input for the second `map` phase is received via the initially created own input channels. The column fragments are composed to form rows of the transposed intermediate result matrix. The second `map f2` application produces the final result of the child processes.

Experimental Results. The following traces and runtime measurements have been obtained on a Beowulf cluster at Heriot-Watt-University, Edinburgh, which consists of 32 Intel Pentium 4 SMP processors running at 3 GHz with 512 MB RAM and a Fast Ethernet interconnection. We implemented the FFT version of Gorlatch and Bischof [7] using our map-and-transpose skeleton. Result collection and post processing (a simple `shuffle`) have been omitted leaving the result matrix in a distributed column-wise manner. A runtime trace, again for input size 4^{10}, is depicted in Figure 5. The communication provoked by the distributed transpose phase overlaps the second computation phase, such that stream communication and computation terminate almost at the same time. The first computation phase is dominant because of the preprocessing, in particular the reordering (bit reversal) of the input list and the computation of the twiddle-factors. Noticeable are also the frequent “runnable” phases, which are garbage collections.

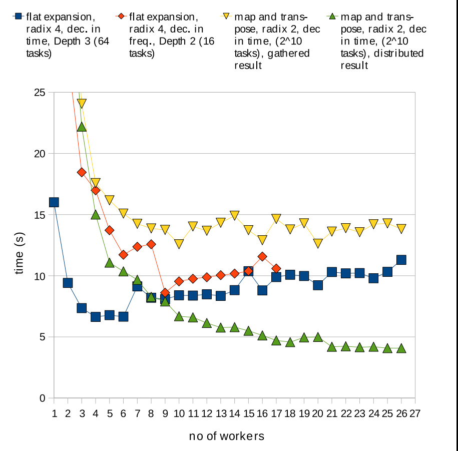
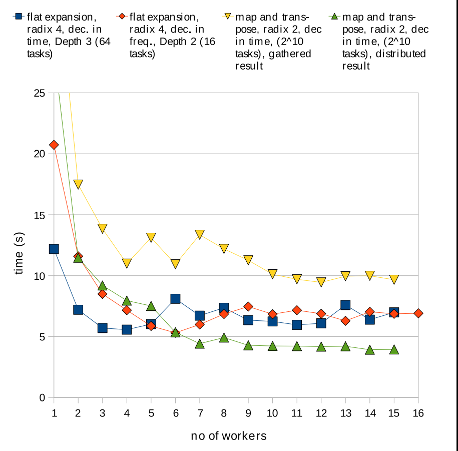
on 22 Pentium 4 CPU's @ 3.00 GHz, 512 MB RAM, fast Ethernet	on 7 Core 2 Duo CPU's @ 2.40 GHz, 2 GB RAM, fast Ethernet
	

Fig. 6. Runtime and scalability comparison of parallel FFT approaches

Figure [6] shows the runtimes of the parallel map-and-transpose FFT version with and without final result collection (Figure [6], triangle marks) in comparison with the best divide-and-conquer versions (4-radix[4], Flat Expansion, Decimation in Time and Frequency). We have measured these versions on the Beowulf cluster and on our local network of dual-core machines, which are more powerful and have more RAM than the Beowulf nodes. The parallel map-and-transpose versions scale well when increasing the number of processing elements. However, for a small number of PEs it is less efficient than the divide-and-conquer versions discussed in Section [2]. Including result collection in the map-and-transpose version decreases the performance clearly. The runtime differences of the various versions are less distinct on the powerful dual-core processors than on the Beowulf nodes. The huge performance penalties of the algorithms with a small number of worker processes on the Beowulf are due to more garbage collection rounds, because of the limited memory size.

4 Related Work

A range of parallel FFT implementations have been presented in the past ([6,5], to mention only a few). The vast majority is tailored for shared-memory systems, see e. g. [9] as an example for a high-level implementation in the functional array language SAC or [1] for experiments with our divide-and-conquer skeletons on multi-core machines. Distributed implementations are mostly based on C+MPI. The distributed MPI-based FFTW implementation [6] is especially tailored for

[4] 4-radix divides the input into 4 parts instead of two.

transforming arrays so large that they do not fit into the memory of a single processor. In contrast to these specialised approaches, our work propagates a skeleton-based parallelisation. In his PhD thesis [11], Christoph Herrmann gives a broad overview, classification, and a vast amount of implementation variants for divide-and-conquer, while we have focused on divide-and-conquer schemes with a fixed branching degree. The skeleton-based version of parallel FFT in [8,7] underlies our parallel map-and-transpose implementation of FFT.

5 Conclusions

The skeleton approach to the parallelisation of FFT provides a high flexibility. In total, six different parallel FFT approaches have been compared, on the basis of three different skeletons: two parallel divide-and-conquer and a parallel map-and-transpose skeleton. We have achieved an acceptable parallel runtime behaviour with a low parallelisation effort. The most effective techniques to lower the communication overhead have been the use of direct mapping to avoid input communication and leaving the results in a distributed manner to avoid the result communication. When applicable, these techniques substantially improve the efficiency.

Acknowledgements. Thanks go to Thomas Horstmeyer for his comments.

References

1. Berthold, J., Dieterle, M., Lobachev, O., Loogen, R.: Distributed memory programming on many-cores – a case study using Eden divide-&-conquer skeletons. In: ARCS Workshop on Many–Cores, Delft, NL, pp. 47–55. VDE–Verlag (2009)
2. Berthold, J., Loogen, R.: Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. In: Proc. of the Intl. Conf. ParCo 2007 – Parallel Computing: Architectures, Algorithms and Applications. IOS Press, Amsterdam (2007)
3. Cole, M.I.: Algorithmic skeletons: Structured management of parallel computation. In: Research Monographs in Parallel and Distributed Computing. Pitman (1989)
4. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Math. Comput. 19, 297–301 (1965)
5. Dmitruk, P., Wang, L., Matthaeus, W., Zhang, R., Seckel, D.: Scalable parallel fft for spectral simulations on a beowulf cluster. Parallel Computing 27(14) (2001)
6. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. of the IEEE 93(2) (2005)
7. Gorlatch, S.: Programming with divide-and-conquer skeletons: A case study of FFT. J. of Supercomputing, 85–97 (1998)
8. Gorlatch, S., Bischof, H.: A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. Par. Proc. Let. 8(4) (1998)
9. Grelck, C., Scholz, S.-B.: Towards an efficient functional implementation of the nas benchmark ft. In: Malyshkin, V.E. (ed.) PaCT 2003. LNCS, vol. 2763, pp. 230–235. Springer, Heidelberg (2003)

10. Gupta, S.K.S., Huang, C.-H., Sadayappan, P., Johnson, R.W.: Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions. Par. Proc. Let. 4(4), 477–488 (1994)
11. Herrmann, C.A.: The Skeleton-Based Parallelization of Divide-and-Conquer Recursions. PhD thesis, Universität Passau (2000) ISBN 3-89722-556-5
12. Klusik, U., Loogen, R., Priebe, S.: Controlling Parallelism and Data Distribution in Eden. In: TFP, vol. 2, pp. 53–64. Intellect (2000)
13. Lobachev, O., Loogen, R.: Towards an Implementation of a Computer Algebra System in a Functional Language. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 141–154. Springer, Heidelberg (2008)
14. Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S., Rubio, F.: Parallelism Abstractions in Eden. In: Rabhi, F.A., Gorlatch, S. (eds.) Patterns and Skeletons for Parallel and Distributed Computing. Springer, Heidelberg (2003)
15. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. J. of Functional Programming 15(3), 431–475 (2005)
16. Nussbaumer, H.: Fast Fourier Transform and Convolution Algorithms. Springer, Berlin (1981)
17. Pease, M.C.: An adaptation of the fast Fourier transform for parallel processing. JACM 15(2), 252–264 (1962)

Parallel Implementation of Generalized Newton Method for Solving Large-Scale LP Problems

Yu.G. Evtushenko, V.A. Garanzha, A.I. Golikov, and H.M. Nguyen

Computing Center RAS, Moscow 119333 Vavilov str. 40, Russia

Abstract. The augmented Lagrangian and Generalized Newton methods are used to simultaneously solve the primal and dual linear programming (LP) problems. We propose parallel implementation of the method to solve the primal linear programming problem with very large number ($\approx 2 \cdot 10^6$) of nonnegative variables and a large ($\approx 2 \cdot 10^5$) number of equality type constraints.

Keywords: linear programming, Newton method, parallel computing.

1 Introduction

In [1], [2] we proposed to use an approach close to the augmented Lagrangian technique. The approach involved has the following main advantage: after a single unconstrained maximization of an auxiliary function which is similar to the augmented Lagrangian we obtain the exact projection of a point onto the solution set of primal LP problem. The auxiliary function has a parameter (similar to the penalty coefficient) which must exceed or be equal to some threshold value. This value is found under the regularity condition (see Theorem 1). Using this result, we maximize once more the auxiliary function with changed Lagrangian multipliers and obtain the exact solution of the dual LP problem (Theorem 2). Theorem 3 states that the exact primal and dual solutions of the LP problem can be obtained in a finite number of iterations with *an arbitrary* positive value of the parameter. The auxiliary unconstrained maximization problems are solved by the fast generalized Newton method. The maximized function is piecewise quadratic, concave, differentiable, but not twice differentiable. Therefore, a generalized Hessian of this function is used. The finite global convergence of the generalized Newton method was established in [3]. Our method for solving LP problems implemented in MATLAB [2]. It was highly competitive to some well-known commercial packages and outperformed them in solving LP problems with a very large number ($\approx \cdot 10^6$) of nonnegative variables and a moderate number ($\approx 5 \cdot 10^3$) of equality type constraints [2].

The proposed parallel approach permits us to increase the number of equality type constraints up to $\approx 2 \cdot 10^5$.

2 Finding a Projection onto the Primal Solution Set

Consider the primal linear program in the standard form

$$f_* = \min_{x \in X} c^\top x, \quad X = \{x \in \mathbb{R}^n : Ax = b, \;\; x \geq 0_n\} \tag{P}$$

together with its dual

$$f_* = \max_{u \in U} \; b^\top u, \quad U = \{u \in \mathbb{R}^m \;\; : \;\; A^\top u \leq c\}, \tag{D}$$

where $A \in \mathbb{R}^{m \times n}$, $c \in R^n$, and $b \in R^m$ are given, x is a primal variable and u is a dual variable, 0_i denotes the i-dimensional zero vector.

Assume that the solution set X_* of the primal problem (P) is nonempty, hence the solution set U_* of the dual problem (D) is also nonempty.

Let $\hat{x} \in \mathbb{R}^n$ be an arbitrary vector. Consider the problem of finding least 2-norm projection $\hat{x}_*$ of the point $\hat{x}$ on X_*

$$\begin{gathered} \tfrac{1}{2}\|\hat{x}_* - \hat{x}\|^2 = \min_{x \in X_*} \;\; \tfrac{1}{2}\|x - \hat{x}\|^2, \\ X_* = \{x \in \mathbb{R}^n \;\; : \;\; Ax = b, \; c^\top x = f_*, \; x \geq 0_n\}. \end{gathered} \tag{1}$$

Henceforth $\|a\|$ denotes the Euclidian norm of a vector a.

The solution $\hat{x}_*$ of problem (1) is unique. Let us introduce the Lagrange function for problem (1)

$$L(x, p, \beta, \hat{x}) = \frac{1}{2}\|x - \hat{x}\|^2 + p^\top(b - Ax) + \beta(c^\top x - f_*),$$

where $p \in \mathbb{R}^m$ and $\beta \in \mathbb{R}^1$ are Lagrange multipliers, $\hat{x}$ is considered to be a fixed parameter vector. The dual problem of (1) is

$$\max_{p \in \mathbb{R}^m} \;\; \max_{\beta \in \mathbb{R}^1} \;\; \min_{x \in \mathbb{R}^n_+} \;\; L(x, p, \beta, \hat{x}). \tag{2}$$

The Kuhn-Tucker conditions for problem (1) imply the existence of $p \in \mathbb{R}^m$ and $\beta \in \mathbb{R}^1$ such that

$$\begin{gathered} x - \hat{x} - A^\top p + \beta c \geq 0_n, \; x \geq 0_n, \\ D(x)(x - \hat{x} - A^\top p + \beta c) = 0_n, \; Ax = b, \; c^\top x = f_*, \end{gathered} \tag{3}$$

where $D(z)$ denotes the diagonal matrix whose ith diagonal element is the ith component of the vector z. It is easy to verify that both inequalities in (3) are equivalent to

$$x = (\hat{x} + A^\top p - \beta c)_+, \tag{4}$$

where a_+ denotes the vector a with all the negative components replaced by zeros.

We can say that (4) gives us the solution of the inner minimization problem in (2). By substituting (4) into $L(x, p, \beta, \hat{x})$, we obtain the dual function

$$\tilde{L}(p, \beta, \hat{x}) = b^\top p - \frac{1}{2}\|(\hat{x} + A^\top p - \beta c)_+\|^2 - \beta f_* + \frac{1}{2}\|\hat{x}\|^2.$$

Hence problem (2) is reduced to the solution of the exterior maximization problem

$$\max_{p \in \mathbb{R}^m} \max_{\beta \in \mathbb{R}^1} \tilde{L}(p, \beta, \hat{x}). \tag{5}$$

If the solutions p and β of problem (5) are found, then after substitution p and β into (4) we obtain the projection $\hat{x}_*$ which solves the problem (1).

The optimality conditions for problem (5) are the following

$$\tilde{L}_p(p, \beta, \hat{x}) = b - A(\hat{x} + A^\top p - \beta c)_+ = b - Ax = 0_m,$$

$$\tilde{L}_\beta(p, \beta, \hat{x}) = c^\top(\hat{x} + A^\top p - \beta c)_+ - f_* = c^\top x - f_* = 0,$$

where x is given by (4). These conditions are satisfied if and only if $x \in X_*$ and $x = \hat{x}_*$.

Unfortunately the unconstrained optimization problem (5) contains an unknown value f_*. It is possible to simplify this problem and remove this shortcoming. We show that if the value β is chosen large enough then the maximization over the variable β can be omitted. Instead of (5) we propose to solve the following simplified unconstrained maximization problem

$$\max_{p \in \mathbb{R}^m} S(p, \beta, \hat{x}), \tag{6}$$

where $\hat{x}$ and β are fixed and the function $S(p, \beta, \hat{x})$ is given by

$$S(p, \beta, \hat{x}) = b^\top p - \frac{1}{2}\|(\hat{x} + A^\top p - \beta c)_+\|^2. \tag{7}$$

Without loss of generality, one can assume that the first l components of $\hat{x}_*$ are strictly greater than zero. In accordance with this assumption, we represent vectors $\hat{x}_*$, $\hat{x}$ and c, as well as the matrix A in the form

$$\hat{x}_*^\top = [[\hat{x}_*^l]^\top, [\hat{x}_*^d]^\top], \ \hat{x}^\top = [[\hat{x}^l]^\top, [\hat{x}^d]^\top],$$

$$c^\top = [[c^l]^\top, [c^d]^\top], \ A = [A_l \mid A_d],$$

where $\hat{x}_*^l > 0_l$, $\hat{x}_*^d = 0_d$, $d = n - l$. In accordance with this representation we can write $v_*^\top = [{v_*^l}^\top, {v_*^d}^\top]$. Consider the Kuhn-Tucker optimality conditions for the primal problem (P). Besides the primal feasibility we have a complementarity condition $\hat{x}_*^\top v_* = 0$, where the dual slack $v_* \in \mathbb{R}^n_+$ and the following dual feasibility conditions have the form

$$v_*^l = c^l - A_l^\top u_* = 0_l, \tag{8}$$
$$v_*^d = c^d - A_d^\top u_* \geq 0_d. \tag{9}$$

The necessary and sufficient optimality conditions (the Kuhn-Tacker conditions) for problem (1) can be written in the expanded form

$$\hat{x}^l_* = \hat{x}^l + A_l^\top p - \beta c^l > 0_l, \tag{10}$$

$$\hat{x}^d_* = 0_d, \ \hat{x}^d + A_d^\top p - \beta c^d \leq 0_d, \tag{11}$$

$$A_l \hat{x}^l_* = b, \quad {c^l}^\top \hat{x}^l_* = f_*.$$

From solutions (10)-(11) we find Lagrange multipliers $[p, \beta]$ such that β is minimal, i.e. consider LP problem

$$\beta_* = \inf_{\beta \in \mathbb{R}^1} \inf_{p \in \mathbb{R}^m} \{\beta \ : \ A_l^\top p - \beta c^l = \hat{x}^l_* - \hat{x}^l, \ A_d^\top p - \beta c^d \leq -\hat{x}^d\}. \tag{12}$$

The constraints in (12) are consistent, but goal function β can be unbounded from below. In this case let be $\beta_* = \gamma$ where γ is an arbitrary number.

If equality system in (12) has single valued solution p, then β_* has following form

$$\beta_* = \begin{cases} \max\limits_{i \in \sigma} \ \frac{(\hat{x}^d + A_d^\top (A_l A_l^\top)^{-1} A_l (\hat{x}^l_* - \hat{x}^l))^i}{(v^d_*)^i}, & \text{if } \sigma \neq \emptyset \\ \gamma > -\infty, \quad \text{if } \sigma = \emptyset \end{cases} \tag{13}$$

where $\sigma = \{1 \leq i \leq d \ : \ (v^d_*)^i > 0\}$ and γ is an arbitrary number.

Theorem 1. *Assume that the solution set X_* for problem (P) is nonempty. Then for all $\beta \geq \beta_*$, where β_* is defined by (12) the unique least 2-norm projection $\hat{x}_*$ of a point $\hat{x}$ onto X_* is given by*

$$\hat{x}_* = (\hat{x} + A^\top p(\beta) - \beta c)_+, \tag{14}$$

where $p(\beta)$ is a point attaining the maximum in (6).

If additionally the rank of submatrix A_l corresponding to nonzero components of vector $\hat{x}_*$ is m, then β_* is defined by (13) and exact solution of (D) is

$$u_* = \frac{1}{\beta}(p(\beta) - (A_l A_l^\top)^{-1} A_l (\hat{x}_* - \hat{x}^l)). \tag{15}$$

This Theorem generalizes the results obtained in paper [4] devoted to finding a normal solution to the primal LP problem. It is obvious that the value of β_* defined by (13) may be negative. The corresponding very simple example is given in Ref. [4].

The function $S(p, \beta, \hat{x})$, where $\hat{x} = 0_n$, can be considered as a new asymptotic exterior penalty function of the dual linear program (D) [4]. The point $p(\beta)$ which maximizes $S(p, \beta, \hat{x})$ does not solve the dual LP problem (D) for finite β, but the ratio $p(\beta)/\beta \to u_*$ as $\beta \to \infty$. If $\beta \geq \beta_*$ then formula (14) provides the exact solution $\hat{x}_*$ to problem (1) (the projection of $\hat{x}$ onto the solution set X_* of the original primal linear program (P)) and if $\hat{x} = 0_n$ then we obtain the exact normal solution of (P).

The next Theorem tells us that we can get a solution to problem (D) from the single unconstrained maximization problem (6) if a point $x_* \in X_*$ is known.

Theorem 2. *Assume that the solution set X_* of problem (P) is nonempty. Then for all $\beta > 0$ and $\hat{x} = x_* \in X_*$ an exact solution of the dual problem (D) is given by $u_* = p(\beta)/\beta$, where $p(\beta)$ is a point attaining the maximum of $S(p, \beta, x_*)$.*

Hence, when Theorem 1 is used and the point $\hat{x}_* \in X_*$ is found, then Theorem 2 provides a very effective and simple tool for solving the dual problem (D). An exact solution to (D) can be obtained by only one unconstrained maximizing the function $S(p, \beta, \hat{x}_*)$ with arbitrary $\beta > 0$.

3 Iterative Process for Solving Primal and Dual LP Problems

In this section we are looking for the arbitrary admissible solutions $x_* \in X_*$ and $u_* \in U_*$ instead of the projection $\hat{x}_*$. Due to this simplification the iterative process described below does not require the knowledge of the threshold value β_*.

Function (7) can be considered as an augmented Lagrangian for the linear program (D). Let us introduce the following iterative process (the augmented Lagrangian method for the dual LP problem (D))

$$p_{s+1} \in \arg \max_{p \in \mathbb{R}^m} \{b^\top p - \tfrac{1}{2}\|(x_s + A^T p - \beta c)_+\|^2\} \tag{16}$$

$$x_{s+1} = (x_s + A^T p_{s+1} - \beta c)_+, \tag{17}$$

where x_0 is an arbitrary starting point.

Theorem 3. *Assume that the solution set X_* of problem (P) is nonempty. Then for all $\beta > 0$ and for arbitrary starting point x_0 the iterative process (16)–(17) converges to $x_* \in X_*$ in a finite number of iterations ω. The formula $u_* = p_{\omega+1}/\beta$ gives an exact solution of the dual problem (D).*

Unconstrained maximization in (6) or (16) can be carried out by the conjugate gradient method or by other iterative methods. Following [3] we utilize the generalized Newton method for solving these subproblems. The resulting iterative method can be described as follows.

1. Set $\beta > 0$, specify the initial points x_0 and p_0, specify the tolerances $tol1$ and tol for outer and inner iterations, respectively.
2. Compute the gradient

$$G_k = \frac{\partial}{\partial p} S(p_k, \beta, x_s) = b - A(x_s + A^\top p_k - \beta c)_+, \tag{18}$$

   on each k-th Newton's inner iteration for solving the unconstrained maximization problem (16), s is the number of external iteration.
3. Using the generalized Hessian for (18) define $m \times m$ matrix

$$H_k = \delta I + A D_k A^T \tag{19}$$

where A – the initial $m \times n$ matrix, the diagonal matrix $D_k \in \mathbb{R}^{n \times n}$ is defined as follow:

$$(D_k)^{ii} = \begin{cases} 1 & \text{if } (x_s + A^\top p_k - \beta c)^i > 0 \\ 0 & \text{if } (x_k + A^\top p_k - \beta c)^i \le 0 \end{cases} \tag{20}$$

4. The maximizing direction δp is found as a solution of the linear system

$$H_k \delta p = -G_k. \tag{21}$$

   Since H_k is a symmetric positive matrix we can use the preconditioned conjugate gradient method for solving the linear system (21). The diagonal part of matrix H_k is used as a preconditioner.
5. Define $p_{k+1} = p_k - \tau_k \delta p$, where τ_k is the stepsize chosen by Armijo rule:

$$\tau_k = \max_{\tau} S(p_k - \tau \delta p, \beta, x_s)$$

   In practice, the parameter τ_k was set as 1 for all taken experiments.
6. If inequality $||p_{k+1} - p_k|| \le tol$ holds then we set $\tilde{p} = p_{k+1}$, compute $x_{s+1} = (x_s + A^\top \tilde{p} - \beta c)_+$. Otherwise we increment iteration number k and go to step 2.
7. If inequality $||p_{s+1} - p_s|| \le tol1$ holds then $u^\star = \tilde{p}/\beta$ and solution to the primal problem (P) is $x^\star = x_{k+1}$. Else we set $p_0 = \tilde{p}$, increment iteration number s and go to step 2.

Distributed memory parallel implementation of the above iterative scheme based on MPI library calls for communications is described below. The basic parallel operations of the algorithm 1-7 which cannot avoid data exchange between nodes are the following:

– multiplication of matrix $A \in \mathbb{R}^{m \times n}$ or $A^\top$ by a vector;
– scalar product of vectors with size n or m;
– forming matrix (19);
– multiplication of matrix (19) by a vector;

For example, the conjugate gradient method requires distributed inner product computations and distributed multiplication of matrix (19) by a vector. Other operations are local. Calculation of gradient (18) requires distributed matrix-vector products with matrices A and A^T.

Efficiency of parallel algorithm crucially depends on problem data distribution across computing nodes. Parallel data partitioning for LP solvers is discussed in [5], [6]. Below we describe and compare several data partitioning schemes.

Block Column Partitioning Scheme. Since number of columns n in matrix A is much larger compared to the number of rows m, we can use simply block column matrix partitioning (“column scheme” for short). Matrix A is split into n_p block column submatrices A_i with approximately the same size. Matrix A_i is stored at the i-th node, as shown in fig. 1.

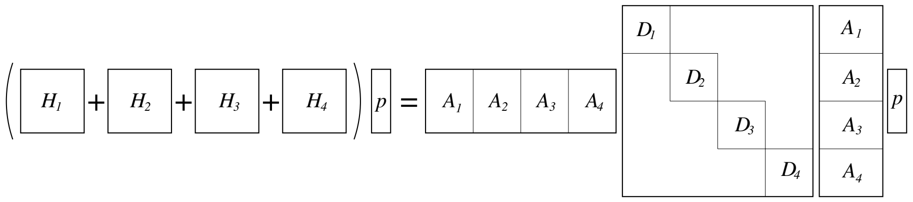


Fig. 1. Block column partitioning scheme

In what follows, for the sake of brevity, we will assume that all submatrices have the same size, i.e. matrix $A_i \in \mathbb{R}^{m \times N_c}$ and subvector $x_i \in \mathbb{R}^{N_c}$, where $N_c = n/n_p$ belongs to i-th processor. Copies of vectors p and b are stored on each processor.

Obviously matrix-vector product $A_i^\top p$ does not require any communications. Otherwise, matrix-vector product Ax can be written as

$$Ax = \sum_{i=1}^{n_p} A_i x_i$$

where i-th processor computes vector $A_i x_i$. The sum of n_p vectors of size m is computed using MPI function `MPI_Allreduce`.

Matrix H can be represented as a sum of n_p terms

$$H = \sum_{i=1}^{n_p} H_i, \text{ where } H_i = A_i D_i A_i^\top.$$

Here $D_i \in \mathbb{R}^{N_c \times N_c}$ are the diagonal blocks of matrix D. In column scheme matrix H is not computed, instead each processor compute independently only its term H_i. As a result the matrix-vector product looks as follow:

$$Hq = \sum_{i=1}^{n_p} H_i q,$$

i.e. vector $H_i q$ is computed on i-th processors and the sum of n_p resulting vectors is computed using MPI function `MPI_Allreduce`. The resulting vector will be available on each processor.

Thus the conjugate gradient solver for linear system (21) is not parallel at all and total computational expences for this solver are proportional to the number of processors.

Let γ denote the ratio of computational expenses for solving linear system to the remaining computational expenses at one iteration of Newton method. If we neglect the time for communications, then the upper bound for parallel speedup can be estimated using simple formula

$$s(n_p) = n_p \frac{1+\gamma}{1+\gamma n}. \tag{22}$$

Thus, if $\gamma = 0.05$ then $s(4) = 3.5$ and $s(6) = 4.85$. If we take into consideration the real communication expenses the speedup could be noticeably worse. Obviously column scheme is effective only in case of sufficiently small γ.

Block Row Partitioning Scheme. In order construct parallel solver for linear system (21) one can use block row partitioning scheme (for short "row scheme") shown on fig. 2.

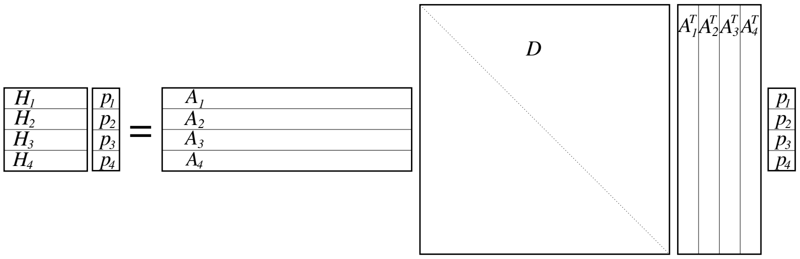


Fig. 2. Block row partitioning scheme

Here matrix A is divided into n_p row blocks. For the sake of brevity we again assume that all blocks contain the same number of rows N_r, thus $m = n_p \times N_r$. In row scheme vector $x \in \mathbb{R}^n$ is copied onto all n_p processors, vector $p \in \mathbb{R}^m$ is distributed and consists of subvectors p_i of the size N_r.

Matrix-vector product $A^T p$ can be written as

$$A^T p = \sum_{i=1}^{n_p} A_i^T p_i. \tag{23}$$

At the first glance, the calculations of i-th term are local and in order to compute $A^T p$ one have to sum up n_p vectors of size n using, for example, function `MPI_Allreduce`. However, the value n appears to be so big that the time spent on communication turns out unacceptably large and does not allow to achieve the speedup.

We face another difficulty while computing the Hessian matrix. If ith processor stores only the submatrix A_i then computing ADA^T will require exchanging all the submatrixes resulting in prohibiting amount of communications.

In order to resolve above problems we store whole matrix A on each node. As a result forming generalized Hessian matrix via

$$H_i = A_i D A_i^T \tag{24}$$

does not require any communications. The floating point operation count at this stage can be estimated as $N_r m \rho^2 \rho_z n$, where ρ and ρ_z are the fractions of nonzero elements in A and in diagonal matrix D, respectively.

In order to compute $A^T p$ the subvectors p_i are gathered into single vector p simultaneously stored on all processors. Thus we avoid $O(n)$ communications, but matrix-vector product is no longer parallel.

Let us introduce the coefficient

$$\alpha = \frac{\rho mn}{m^2 n \rho^2 \rho_z} = \frac{1}{m\rho\rho_z} \tag{25}$$

being an approximate ratio of computational cost for multiplying matrix A by a vector to expenses for forming Hessian matrix in serial implementation. Using this coefficient one can easily derive crude but simple upper bound of speedup for row scheme:

$$s(n_p) = n_p \frac{1 + 2\alpha}{1 + (n_p + 1)\alpha} \tag{26}$$

Numerical experiments show that formula (25) is too pessimistic and real ratio is much smaller. If we set $\alpha = 0.1$ then $s(8) = 5.05$.

The main drawback of the row scheme is that it is not memory parallel since leading term of the total memory estimate looks as ρmnn_p and is not scaled with the number of processors.

On the bright side, if the matrix A is supersparse in a sense that the matrix H is also sparse, then the block row partitioning scheme can be very efficient but parallel algorithm should be modified. In particular, the row scheme allows highly effective parallel implementation of the preconditioned conjugate gradient method for solving linear system, the reliable partial LU-decomposition can be use as a preconditioner. Thus row scheme based algorithm can be used for very stiff LP problems.

Matrix-Free Algorithm. The memory nonoptimality of row scheme sharply limits the maximum size of solvable problems, so there arises the necessity of building the algorithm that is memory optimal and allows to keep only one block row of matrix A per processor.

We implemented the matrix-free version of the Newton method where the Hessian matrix H is not formed at all, we only compute it's main diagonal. Matrix-vector product $q = Ap$ can be computed as a sequence of operations

$$q_i = A_i \sum_{j=1}^{n_p} DA_j^T p_j,$$

where subvector p_j is stored at j-th processor. In the simplest implementation the term $DA_i^T p_i$ can be computed by i-th processor, while the sum of n_p terms can be computed using function `MPI_Allreduce`. The resulting vector can be multiplied by matrix A_i. In order to avoid $O(n)$ data exchanges we use the sparseness of diagonal matrix D, which essentially means that the number of nonzeros in vector $DA_j^T p_j$ cannot exceed number of nonzeros in D. Thus the number of nonzeros in sparse vectors can be estimated as $\rho_z n$ which can be hundred times smaller compared to n. Sparse vectors are packed and sent to another processors. Note that integer sparseness data should be sent as well.

Unfortunately, on the stage of forming matrix D one still need to compute A^Tp, therefore we still have to sum n_p vectors of the size n at least once at each Newton iteration. Thus we do not expect to receive any speedup for resulting matrix-free algorithm, we rather hope that it's parallel implementation would not be slower than the sequential one. In this case the parallel implementation will be much more effective compared to out-of-core LP solvers.

General Block Partitioning Scheme. In order to attain higher parallel efficiency it is reasonable to use general block partitioning scheme ("cellular scheme" for short), which is the combination of column and row schemes, with matrix A divided into rectangular blocks as shown on fig.3.

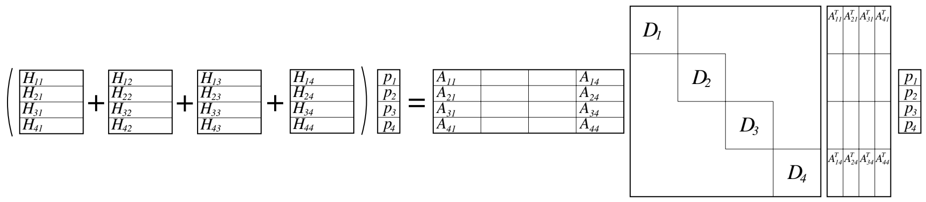


Fig. 3. Cellular scheme

We assume that the total number of processors can be presented as $n_p = n_r \times n_c$, e.g. the processors are placed at nodes of the $n_r \times n_c$ grid. For the brevity sake we assume again that $n = n_c \times N_c$ $m = n_r \times N_r$. Matrix A consists of n_p submatrixes $A_{ij} \in \mathbb{R}^{N_r \times N_c}$. In this scheme vector $x \in \mathbb{R}^n$ is divided into n_c subvectors x_i, vector p is divided into n_r subvectors p_j, subvector x_i is stored on each processor of the i-row of the grid simultaneously and subvector p_j is copied on all n_c processors of j-th column of the grid. As in row scheme, we assume that for all i, the i-th processor of jth column in the grid keeps the whole j-th column block of the matrix A, and not just the submatrix A_{ij}. Thus we store n_r copies of matrix A.

Assuming that vector p is available at each of $n_r \times n_c$ processors, the same formula

$$x_i = \sum_{j=1}^{n_r} A^T_{ji} p_j$$

can be used for all nodes in a i-th column of the grid independently. Thus vector x_i is available on all processors of the i-th column without communications, but matrix A^T is multiplied by vector p n_r times. Thus arithmetic cost of this step is close to ρmnn_r.

It is convenient for implement independent collective operations within one vertical or horizontal line of the processor grid using the communicator splitting function provided by MPI's library.

The operation Ax turns out to be almost local and requires communications only for summing n_c subvectors of size N_r independently on each group of n_c processors in n_r rows of the grid.

The assembling of the Hessian matrix requires only local computations.

Using coefficients γ and α defined as above we can derive the rather crude but simple upper bound for the speedup of cellular scheme:

$$s_u(n_r \times n_c) = n_r \frac{1+2\alpha}{1+(n_r+1)\alpha} n_c \frac{1+\gamma}{1+\gamma n_c} \tag{27}$$

If we set $\gamma = 0.05$ and $\alpha = 0.1$ in formula (27), then $s_u(8 \times 8) = 30.3$.

Cyclic Data Partitioning with Reflections. In cellular scheme we did not take into account the symmetry of generalized Hessian matrix which could cut down the computational cost for forming matrix H almost by half. It is very easy to use symmetry in the column scheme, but the attempt to do the same in the row and cellular schemes leads to load disbalance of processors for forming H as well for computing product of H by a vector.

In order to attain optimal load balancing one can use the well-known cyclic data partitioning with reflections shown on fig. 4 in the case of $n_p = 4$.

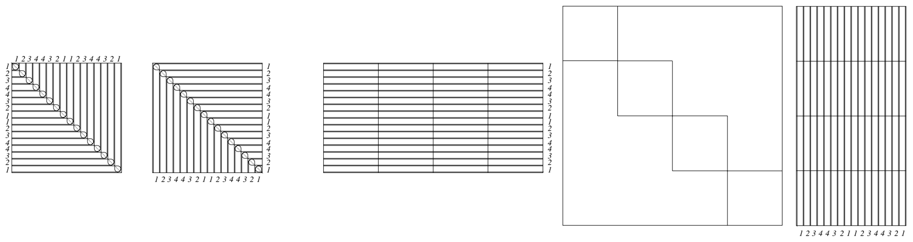


Fig. 4. Cyclic data partitioning with reflections

This partitioning is close to the cellular scheme, but the rows of matrix A are divided into groups, in each group row are distributed between processors, and numbering scheme between nearby groups is changed to opposite (reflected). One can easily check that such a data distribution can balance both symmetric matrix assembly and matrix-vector product.

This scheme was not implemented in current work but simpler cellular scheme was used to model its behaviour. However one should keep in mind that in cyclic scheme coefficient α can be almost two times larger compared to cellular one so speedup results can be somewhat worse.

4 Results of Numerical Experiments

We used synthetic LP test problems [1], [2]. The test problem generator produces for given solutions x_* and u_* of the LP problem a random matrix A for a given

Table 1. Results for the column scheme with $m = 10^4$, $n = 10^6$, $\rho = 0.01$

n_p	1	2	4	8	16	32	64
T_{tot} (sec.)	1439.32	1311.40	568.79	408.94	236.68	188.17	142.65
s_{tot}	1	1.10	2.53	3.52	6.08	7.65	10.09
T_{lin}(sec.)	121.88	124.42	122.09	120.60	116.87	109.80	106.17
s_{lin}	1	0.98	1.00	1.01	1.04	1.11	1.15
T_{rem} (sec.)	1317.35	1186.69	446.61	288.25	119.73	78.30	36.40
s_{rem}	1	1.11	2.95	4.57	11.00	16.82	36.19

Table 2. Results for the row scheme with $m = 5000$, $n = 10^6$, $\rho = 0.01$

n_p	1	2	4	8	16	32	64
T_{tot} (sec.)	406.66	242.78	121.32	62.02	32.13	18.43	15.49
s_{tot}	1	1.67	3.35	6.56	12.66	22.07	26.24
T_{lin}(sec.)	31.47	16.45	8.36	4.4	2.28	1.55	2.09
s_{lin}	1	1.91	3.77	7.16	13.81	20.24	15.04
T_{rem} (sec.)	375.13	226.24	112.88	57.53	29.76	16.79	13.32
s_{rem}	1	1.66	3.32	6.52	12.60	22.35	28.16

Table 3. Results for the cellular scheme with $m = 10^4$, $n = 10^6$, $\rho = 0.01$

$n_p = n_r \times n_c$	$1 = 1 \times 1$	$4 = 2 \times 2$	$16 = 4 \times 4$	$64 = 8 \times 8$
T_{tot} (sec.)	1439.32	502.98	145.13	45.65
s_{tot}	1	2.86	9.92	31.53
T_{lin} (sec.)	121.88	62.17	31.66	15.89
s_{lin}	1	1.96	3.85	7.67
T_{rem} (sec.)	1317.35	440.73	113.43	29.74
s_{rem}	1	2.99	11.61	44.30

Table 4. Results for matrix-free algorithm

$m \times n \times \rho$	n_p	T_{tot}	Δ_1	Δ_2	Δ_3
$5 \cdot 10^4 \times 10^6 \times 0.01$	16	400.02	$2.1 \cdot 10^{-6}$	$2.1 \cdot 10^{-7}$	$1.1 \cdot 10^{-10}$
$10^5 \times 10^6 \times 0.01$	20	484.62	$8.1 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$	$5.6 \cdot 10^{-11}$
$10^5 \times 2 \cdot 10^6 \times 0.01$	40	823.13	$4.5 \cdot 10^{-6}$	$4.2 \cdot 10^{-7}$	$7.2 \cdot 10^{-11}$
$2 \cdot 10^5 \times 2 \cdot 10^6 \times 0.01$	80	2317.42	$4.9 \cdot 10^{-5}$	$6.6 \cdot 10^{-6}$	$5.2 \cdot 10^{-10}$

m (number of equality constraints) and n (number of nonnegative variables) and density ρ. In particular, $\rho = 1$ means that all the entries in A were generated as random numbers, whereas $\rho = 0.01$ indicates that only one percent of the entries in A was generated randomly and others were set equal to zero.

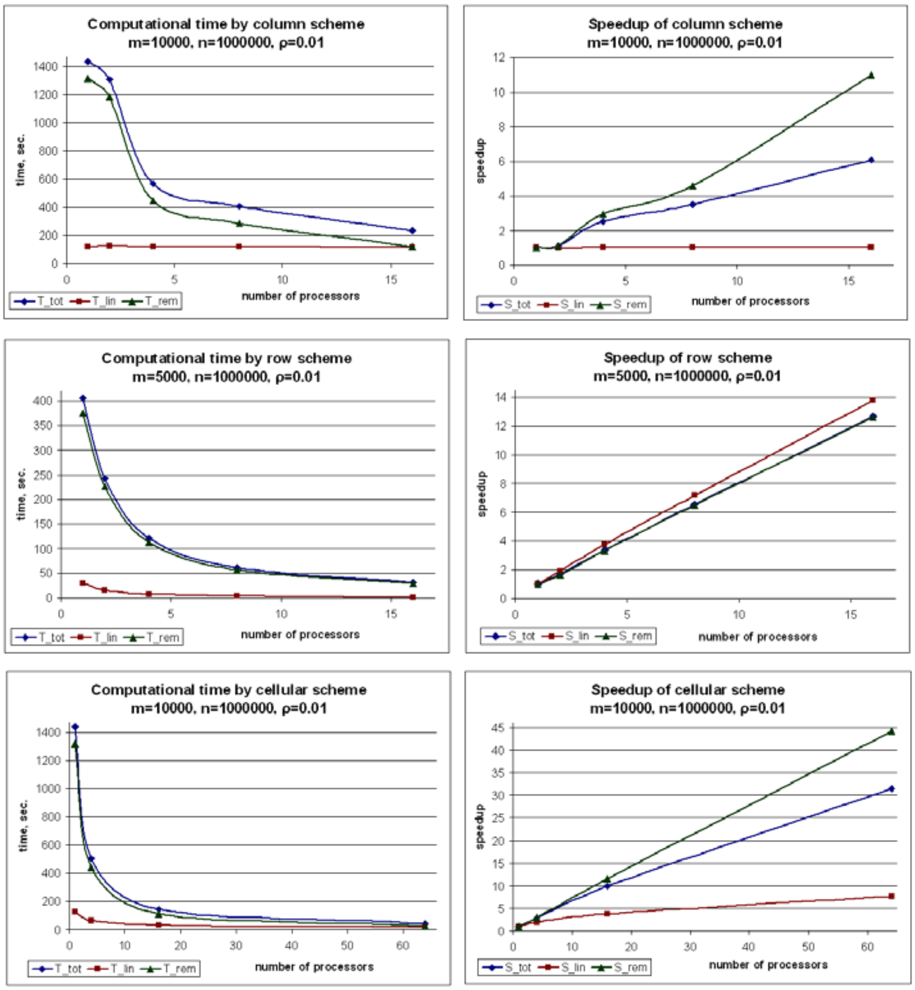


Fig. 5. Speedup diagrams

In tables 1-3 the total wall-clock time for solving LP problem is denoted as T_{tot}, T_{lin} means the time spent on solving linear systems ([21]), T_{rem} is time for remaining computations. The total speedup is denoted as S_{tot}, speedup of solving linear systems is S_{lin} and S_{rem} means the speedup of the remainder.

Table 4 presents some computational results for matrix-free algorithm. Residuals of LP problem are $\Delta_1 = \|Ax-b\|_\infty$, $\Delta_2 = \|(A^T u-c)_+\|_\infty$, $\Delta_3 = |c^T x-b^\top u|$.

Speedup results are illustrated in Fig. 5.

Conclusions and Directions of Further Research. Parallel versions of LP solver based on several data distribution schemes were implemented and applied to large scale LP problems. As expected, constructing efficient parallel algorithm for LP problems was found to be quite hard problem thus in each version of parallel algorithm we were looking for acceptable compromise between arithmetic scalability and memory scalability. In the current setting best speedup results were attained using general block partitioning scheme, however for various combinations of size and sparsity pattern of the matrix A one should look for optimal data distribution scheme between row partitioning, column partitioning and general block partitioning.

While the resulting speedup is acceptable (above 30 on 64 cores), one should keep in mind that in practice creating input data for parallel LP solver is quite nontrivial task and actually it can be more costly compared to parallel solution.

We did not used shared memory parallelism in the current implementation of the numerical algorithm and one can expect that combination of distributed/ shared memory algorithms based on MPI/OpenMP can much better suit to multi-core architecture of modern CPU's.

Another important issue is optimization of operations with dense and sparse matrices. While we have used Lapack functions for efficient computation of matrix-vector product Hp assuming that matrix H is a full matrix, operations with sparse matrices still leave much room for optimization. In particular, one can use special cash-aware or cash-independent sparse matrix storage schemes, potentially may sharply reduce wall-clock time for sparse matrix-vector products and change the speedup results.

Acknowledgments. This study was supported by the Russian Foundation for Basic Research (project no. 08-01-00619) and by the Program of the State Support of Leading Scientific Schools (project no.NSh-5073.2008.1).

References

1. Golikov, A.I., Evtushenko, Yu.G.: Solution Method for Large-Scale Linear Programming Problems. Doklady Mathematics 70(1), 615–619 (2004)
2. Evtushenko, Yu.G., Golikov, A.I., Mollaverdi, N.: Augmented Lagrangian method for large-scale linear programming problems. Optim. Methods and Software 7(4-5), 515–524 (2005)
3. Mangasarian, O.L.: A Newton Method for Linear Programming. Jour. of Optim. Theory and Appl. 121, 1–18 (2004)
4. Golikov, A.I., Evtushenko, Yu.G.: Search for Normal Solutions in Linear Programming Problems. Comput. Math. and Math. Phys. 40, 1694–1714 (2000)
5. Karypis, G., Gupta, A., Kumar, V.: A parallel formulation of interior point algorithms. In: Proceedings of Supercomputing, pp. 204–213 (1994)
6. Coleman, T.F., Czyzyk, J., Sun, C., Wagner, M., Wright, S.J.: pPCx: Parallel Software for Linear Programming. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, Hyatt Regency Minneapolis on Nicollel Mall Hotel, Minneapolis, Minnesota, USA, March 14-17. SIAM, Philadelphia (1997)

Dynamic Real-Time Resource Provisioning for Massively Multiplayer Online Games[⋆]

Radu Prodan, Vlad Nae, Thomas Fahringer, and Herbert Jordan

Institute of Computer Science, University of Innsbruck,
Technikerstraße 21a, A-6020 Innsbruck, Austria
{vlad,radu,tf,jordan}@dps.uibk.ac.at

Abstract. Massively Multiplayer Online Games (MMOG) are a class of computationally-intensive client-server applications with severe real-time Quality of Service (QoS) requirements, such the number of updates per second each client needs to receive from the servers for a fluent and realistic experience. To guarantee the QoS requirements, game providers currently over-provision a large amount of their resources, which makes the overall efficiency of provisioning and utilisation of resources rather low and prohibits any but the largest providers from joining the market.

To address this deficiency, we propose a new prediction-based method for dynamic resource provisioning and scaling of MMOGs in distributed Grid environments. Firstly, a load prediction service anticipates the future game world entity distribution from historical trace data using a fast and flexible neural network-based method. On top of it, we developed generic analytical game load models used to foresee future hot-spots that congest the game servers and make the overall environment fragmented and unplayable. Finally, a resource allocation service performs dynamic load distribution, balancing, and migration of entities that keep the game servers reasonably loaded such that the real-time QoS requirements are maintained.

1 Introduction

Online entertainment including gaming is a huge growth sector worldwide. Massively Multiplayer Online Games (MMOG) grew from 10 thousand subscribers in 1997 to 6.7 million in 2003 and the rate is accelerating estimated to 60 million people by 2011. The release of World of Warcraft in 2005 saw a single game break the barrier of 4 million subscribers worldwide. The market size shows equally impressive numbers, estimated by the Entertainment Software Association (ESA) to 7 billion USD with an avid growth over 300% in the last 10 years. In comparison, the Motion Picture Association of America (MPAA) reports a size of 8.99 billion USD and the Recording Industry Association of America (RIAA) a size

[⋆] This research is partially funded by the European Union through the IST-034601 edutain@grid project

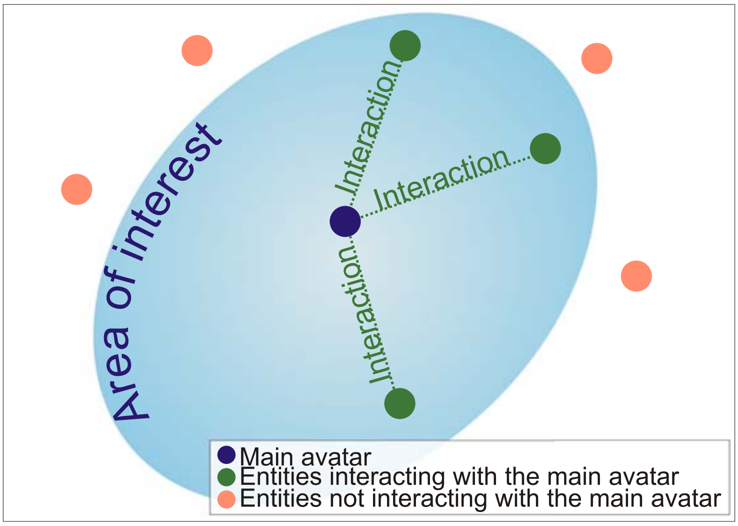

Area of interest
Interaction
Interaction
Interaction
● Main avatar
● Entities interacting with the main avatar
● Entities not interacting with the main avatar

Fig. 1. Area of interest

of 12.3 billion USD which stagnates (and even decreased by 2%) in the last 10 years. It is therefore expected that the game industry will soon grow larger than both movie and music market sizes.

Today's MMOG operate as client-server architectures, in which the game server simulates a world via computing and database operations, receives and processes commands from the clients (shootings, collection of items, chat), and inter-operates with a billing and accounting system. The game severs must respond with new game state information to the distributed clients promptly within a given *real-time* interval to ensure a smooth, responsive, and fair experience for all players. Depending on game, typical response times must be between 100 milliseconds (10 hertz) to ensure fluent play in online First Person Shooter (FPS) action games. Failing to deliver timely simulation updates leads to a degraded game experience and brings unhappiness to players that cancel their accounts.

2 Background

The vast majority of games follow a client-server architecture with a similar computational model. The game server runs an infinite loop in which the state of all entities is computed, stored into a persistent database, and then broadcasted to the clients. All entities within a specific avatar's *area of interest* (usually a surrounding zone) are considered to be interacting with the respective avatar and have an impact on its state. There are four main factors that affect the load of a game session: the size of the game world, the total number of entities, the density of entities within the area of interest, and the level of interaction.

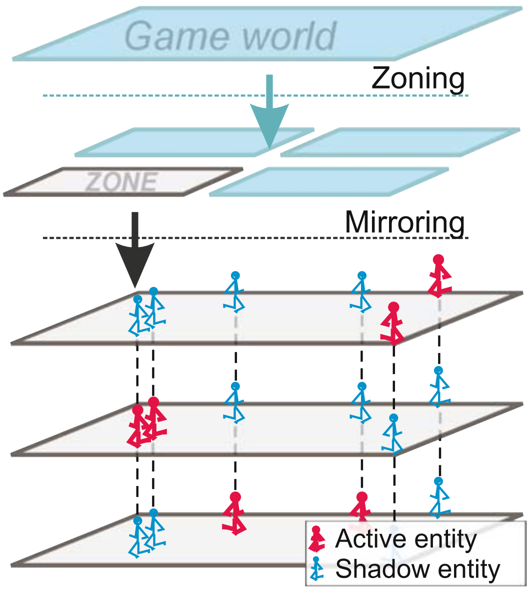


Fig. 2. Zoning and mirroring

Obviously, the more populated the entities' areas of interest are and the more interactions between entities exist, the higher the load of the underlying game server will be. An overloaded game server delivers state updates to clients (i.e. movements and actions of teammates and opponents) at a lower frequency than required which makes the overall environment fragmented and unplayable.

Today, a single computer is limited to around 500 simultaneous and persistent network connections, and databases can manage the update of around 500 objects per second [10]. To support at the same time millions of active concurrent players and many more other game entities, Massively Multiplayer Online Games (MMOG) operators are distributing the load of a game world across multiple computational resources using three main techniques: zoning, mirroring, and instancing.

Spatial scaling of a game session is achieved through a conventional parallelization technique called *zoning* [3], based on similar data locality concepts as in scientific parallel processing. Zoning partitions the game world into geographical areas to be handled independently by separate machines (see Figure 2). Zones are not necessarily of same shape and size, but should have an even load distribution that satisfies the Quality of Service (QoS) requirements. Today, zoning is successfully employed in slower-paced (compared fast-paced to FPS action games) adventure games, widely known as Massively Multiplayer Online Role Playing Games (MMORPG) [9], where the transition between zones can only happen through certain portals (e.g. special doors, teleportation, accompanied on the screen by a load clock or some standard animation video) and requires an important amount of time. Typically, zones are started manually by the game operators based on the current load, player demand, or new game world and scenario developments.

The second technique called *mirroring* [8] targets parallelization of game sessions with a large density of players located and interacting within each other's area of interest (see Figure 2). Such situations are typical to fast-paced FPS action games in which players typically gather in certain hot-spot action areas that congest the game servers that are no longer capable of delivering state updates at the required rate. To address this problem, mirroring defines a novel method of distributing the load by replicating the same game zone on several CPUs. Each replicated server computes the state for a subset of entities called *active entities*, while the remaining ones, called *shadow entities* (which are active in the other participating servers), are synchronised across servers. It was proven in previous research that the overhead of synchronising shadow entities is much lower than the overhead of computing the load produced by active entities [8].

The third technique called *instancing* is a simplification of mirroring which distributes the session load by starting multiple parallel instances of the highly populated zones. The instances are completely independent of each other, which mean that two avatars from different instances will not see each other, even if they are located at coordinates within their area of interest. This technique is relatively easy to implement based on the zoning technique and is mostly employed for MMORPGs, where the implementation "cheat" of starting multiple independent instances of the same zone is less visible to the players.

Work at the University of Münster is developing the Real-Time Framework (RTF) [6] that proposes to the game developers a portable API and optimised protocols which facilitate parallelization of game sessions using the zoning, mirroring, and instancing techniques.

3 Method

To accommodate such a huge user load (millions of players), the game providers typically install and operate a large infrastructure, with hundreds to thousands of computers for each game in order to provide the required QoS. For example, the operating infrastructure of the MMOG World of Warcraft [2] has over 10 thousand computers. However, similar to fashion goods, the demand of a MMOG is highly dynamic and thus, even for the large providers that operate several titles in parallel, a large portion of the resources are unnecessary which leads to a very inefficient and low resource utilisation. In addition, this enterprise limitation has negative economic impacts by preventing any but the largest hosting centres from joining the market which will dramatically increase prices because those centres must be capable of handling peaks in demand, even if the resources are not needed for much of the time.

To alleviate this problem, we propose to use the Grid computing potential of providing on-demand access to the huge amount of cheap computers connected to the Internet. Despite this advantage, delivering the required real-time

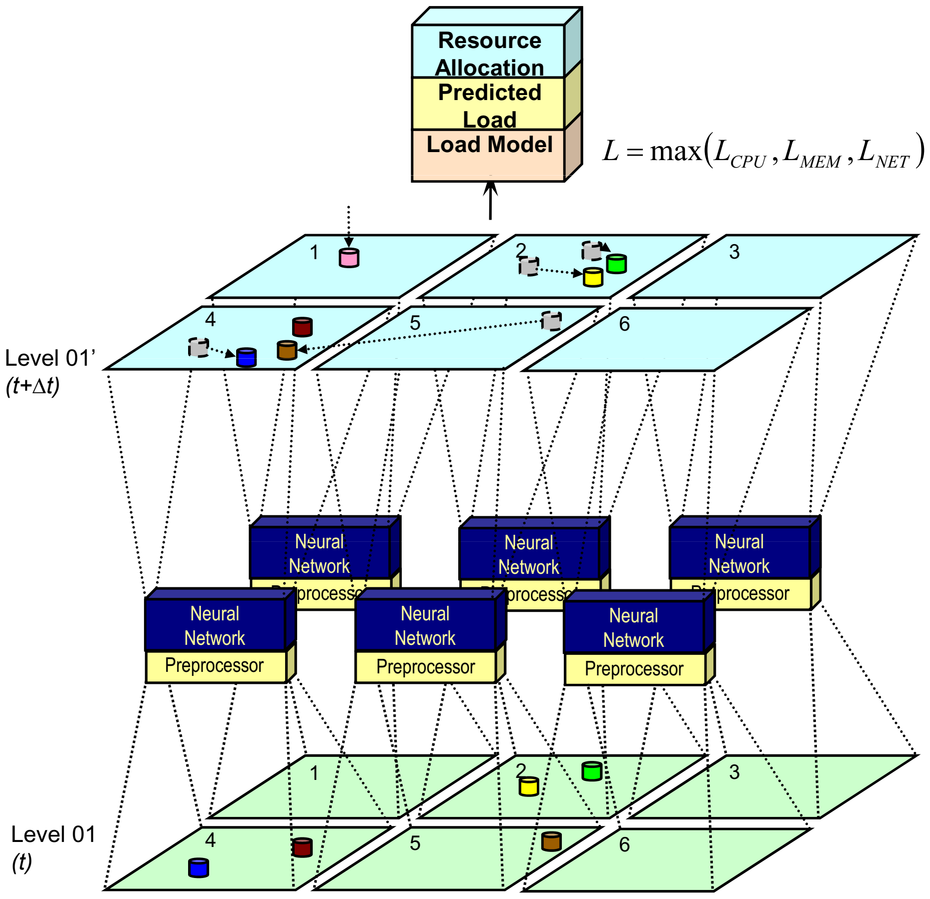


Fig. 3. Prediction-based resource provisioning method

QoS needs remains a challenging task since MMOGs, and especially FPS games, are highly dynamic and allow many users concentrate in each other's proximity within a short period of time causing excessive server load that no longer sends state updates at the required rate. Dynamically deciding and establishing a new parallelization strategy like zoning, mirroring, and instances may be under circumstances an expensive operation taking several seconds that causes unacceptable delays in the users' experience if not hidden properly by the middleware.

To solve this challenge, we designed a resource allocation and provisioning method consisting of three services (see Figure 3). A *load prediction* service sketched in Section 4 is in charge of projecting the future distribution of entities in the game world and tries to timely foresee critical hot-spots that congest the game servers. On top of it, a *load modelling* service described in detail in Section 5 is using analytical methods for estimating the game load based on entity distribution and possible interactions. Finally, a *resource allocation* service summarised in Section 6 is using the load information to trigger new game distributions through zoning, replication, or instancing, that accommodate the player load while guaranteeing the real-time QoS constraints.

We present experimental results that validate our methods in Section 7 and conclude in Section 8.

4 Load Prediction

The load of MMOGs is highly dynamic not only because of the high number of players connected to the same game session, but also due to their often unpredictable interactions. The interaction between players depends on their position in the game world and on whether they find themselves in each other's area of interest. Ultimately, the load of a game session depends therefore on the position of players in the game world which is the task of the load prediction service.

Highly dynamic real-tome applications like online games require fast prediction methods in order to be of any real use. At the same time, the (often unpredictable) human factor makes the problem even harder and requires adaptive techniques, which simple methods such as average, exponential smoothing, or last value fail to achieve. We therefore decided on a solution based on neural networks due to a number of reasons that make them suitable for predicting the load of online game sessions in real-time, as we will experimentally demonstrate in Section 7.1: they adapt to a wide range of time series, they offer better prediction results than other simple methods, and they are sufficiently fast compared to other more sophisticated statistical analysis.

Our neural network-based prediction strategy is to partition the game world into subareas, where the size of a subarea needs to be small enough such that its load can be characterised by the entity count. The overall entity distribution in the entire game world consists of a map of entity counts for each subarea. The predictor uses one separate neural network for each subarea which receives as input the entity count at equidistant past time intervals and delivers as output the entity count at the next time step (see Figure 3). The predicted entity count for the entire game world is the sum of all the subarea predictions, which will be used afterwards by the load modelling service for estimating the game server load.

5 Load Modelling

Having the future entity distribution information produced by the load prediction service, the goal of load modelling is to perform the mapping to machine load information. As part of this service, we propose analytical models for expressing the load of three type of resources that are mostly used by the current MMOGs: CPU, memory, and network.

Let us consider N clients connected to a distributed game session aggregating a total of H machines from different resource providers. Let us further consider that inside the game world roam BE client-independent entities. On each the machine, only AE entities are active (not shadows) and there are C clients connected (see Section 2).

The game intelligence consists of a single game loop, which is invariably true for all modern games. A game loop iteration is called a *tick* and in each tick there are certain steps that have to be performed: (i) processing events coming from the clients connected to the current machine; (ii) processing state updates received from the other machines for the shadow entities (client-independent and otherwise); and (iii) processing the states of the active entities (client-independent and otherwise).

5.1 CPU Load Model

In this section we propose an analytical model for the load of one single machine in a distributed game session.

We can distinguish three basic time consuming activities within one game tick: (i) the computation of an interaction between two entities t_i; (ii) the receipt of an event message from one client t_m; and (iii) the update of one entity's state received/sent from/to another machine t_u. In order to keep the complexity of this model acceptable, we assume that user-independent entities do not interact among themselves, which is true in the majority of cases. We model the CPU time t_M spent for sending and receiving messages from each client to a server (active client-controlled entities) as follows:

$$t_M = C \cdot t_m.$$

The CPU time t_U spent by the server for processing the state updates from the other machines is:

$$t_U = (N - C) \cdot t_u + (BE - AE) \cdot t_u + AE \cdot t_u,$$

and the CPU time t_I spent by the server for computing the interactions that involve active entities is:

$$t_I = I \cdot t_i,$$

where I is the total number of interactions involving the active entities. Obviously, the computation of interactions that do not involve active entities is allotted to other machines.

Let us denote by IC the number of interacting client-controlled entities with any other entities (client-controlled or otherwise). Furthermore, we define p_{ci} as the average number of interactions involving active client-controlled entities expressed as a percentage of IC. Analogously, we define p_{ei} as the average number of interactions involving active client-independent entities expressed as a percentage of BE. The total number of interactions will composed of the number of interactions between active client-controlled entities and the number of interactions between active client-controlled and client-independent entities:

$$I = p_{ci} \cdot IC^2 + p_{ei} \cdot IC \cdot BE.$$

Consequently, the CPU time t_I for processing the interactions involving all active entities can be calculated as follows:

$$t_I = \big(p_{ci} \cdot IC^2 + p_{ei} \cdot IC \cdot BE\big) \cdot t_i.$$

Approximating the time consumed for sending/receiving an event message as equal to the time needed to update the state of one entity ($t_m = t_u$), the total CPU time consumed in one tick becomes:

$$t_C = (N + BE) \cdot t_u + \big(p_{ci} \cdot IC^2 + p_{ei} \cdot IC \cdot BE\big) \cdot t_i.$$

Furthermore, quantifying t_i with regard to t_u as $t_i = p_{ui} \cdot t_u$, the CPU time consumed in one tick becomes:

$$t_C = \big(N + BE + p_{ui} \cdot p_{ci} \cdot IC^2 + p_{ui} \cdot p_{ei} \cdot IC \cdot BE\big) \cdot t_i.$$

Finally, considering t_{SAT} as the tick saturation threshold, we can define the CPU load function as follows:

$$L_{CPU} = \frac{t_C}{t_{SAT}} = \frac{N + BE + p_{ui} \cdot p_{ci} \cdot IC^2 + p_{ui} \cdot p_{ei} \cdot IC \cdot BE}{v},$$

where v is the CPU speed expressed as an integer representing the number of t_u-long tasks the CPU is able to perform in a t_{SAT}-long time interval.

5.2 Memory Load Model

The memory model is less complex than the processor load model, since all machines involved in the mirroring process keep the entity-state records for all entities participating in the game session. First, we take into account the game-dependent constants such as the amount of memory m_{game} needed to run the actual game engine with no game world loaded and no clients connected. Next, we define m_{world} as the amount of memory used for the game world being played. As for entity-related memory constants, let m_{cs} denote the amount of memory needed to store the state of one client-controlled entity, and m_{es} the amount of memory needed to store the state of a client-independent entity. We ignore the interaction between entities because it does not have a significant impact on the memory load. Aggregating all the data, the memory consumption M on a machine taking part in a distributed game session is:

$$M = N \cdot m_{cs} + BE \cdot m_{es} + m_{game} + m_{world}.$$

As a consequence, the final memory load function is:

$$L_{MEM} = \frac{M}{M_{machine}} = \frac{N \cdot m_{cs} + BE \cdot m_{es} + m_{game} + m_{world}}{M_{machine}},$$

where $M_{machine}$ represents the amount of memory available on the respective machine.

5.3 Network Load Model

In terms of network consumption, we define first the outgoing network bandwidth usage for a machine running a zone of a distributed game session as follows:

$$D_{out} = C \cdot d_{cout} + (H - 1) \cdot (C + AE) \cdot d_{update},$$

where d_{cout} represents the amount of data sent to a client and d_{update} the amount of data exchanged between machines for updating a single entity's state.

Secondly, the incoming network bandwidth usage for a machine running a zone of a distributed game session is defined as:

$$D_{in} = C \cdot d_{cin} + (N - C + BE - AE) \cdot d_{update},$$

where d_{cin} represents the amount of data received from a client.

5.4 Overall Load Model

We merge the previously presented models into an overall resource load model for MMOGs, where the load of the entire system is imposed by the maximum load of the individual resources:

$$L = \max\left(L_{CPU}, L_{MEM}, L_{NET}\right).$$

To stress the generality of our modelling approach, MMOG classes can be defined using the set of constants involved in all of the models previously described:

$$MMOG_{class} = \left\{(BE), (m_{cs}, m_{es}, m_{game}, m_{world}), (d_{cout}, d_{cin}, d_{update})\right\}.$$

Obviously, BE is not MMOG-dependent, but rather game world and game play style (e.g. single play, team play)-dependent. Nevertheless, we included it among the constants defining the MMOG class because we can consider games running different game worlds and game play styles as belonging to different MMOG classes.

6 Resource Allocation

Based on the predicted resource load within the next time interval, the resource allocation service arranges for the provisioning of the resources required for a proper execution that guarantees a good experience to all players. A typical action performed by the resource allocation service is to extend game sessions with new zones or replication servers to accommodate an increased number of players during peak hours. Conversely, the resource allocation service deallocates and merges multiple under-utilised game servers to improve the resource utilisation. The resources that need to be provisioned by the resource allocation service can be of four types, as considered by the load modelling service: CPU, memory, input from the external network, and output to the external network of a resource provider.

After allocating the required resources, the resource allocation service instructs the game servers through the RTF API [6] what parallelization strategy to apply and which entities to migrate to new servers (see Section 2). Since the allocation of resources and establishing of a new game session load distribution scheme is a latency prone task (several seconds), an important aspect is to trigger it early enough using load prediction and modelling services such that the users to not experience any lags during their play.

An important remaining aspect is that resource providers use in general different policies describing one *time bulk* and one *resource bulk* as the minimum allocation units for each type of resource. The measurement unit for the policy resources is a generic "unit'" which represents the requirement for the respective resource from a fully loaded game server (e.g. one CPU unit represents the CPU demand for a fully loaded game zone).

7 Experiments

We present in this section experimental results that validate our load prediction and resource allocation approaches.

7.1 Load Prediction

As there is no real online game available which is capable of generating the load patterns required for a thorough testing and validation of our prediction method, we developed a distributed FPS game simulator on top of the RTF library [6] supporting the zoning technique and the inter-zone migration of entities

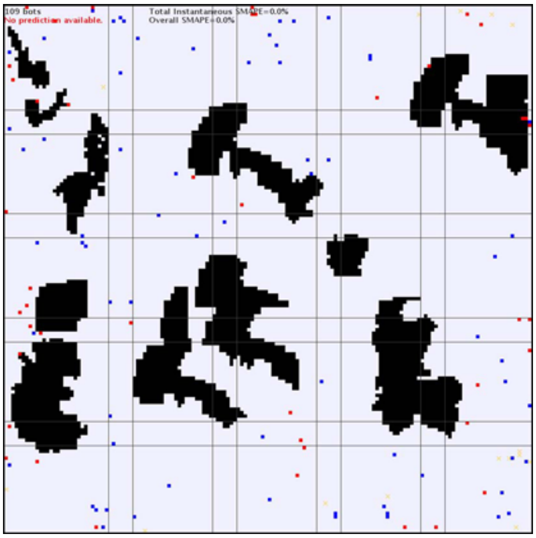


Fig. 4. Game simulator snapshot

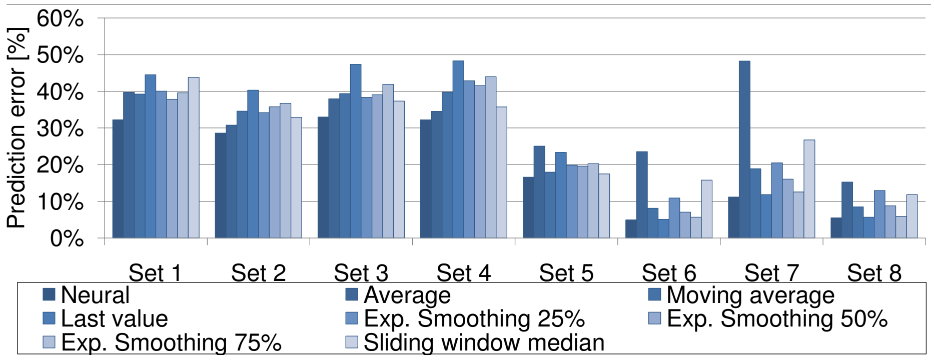
Table 1. Simulation trace data sets

Data set	Peak hours modelling	Peak load	Overall dynamics (17 h.)	Instantaneous dynamics (2 min.)
Set 1	No	+++	+++++	+++
Set 2	No	+	+++	++++
Set 3	No	++++	++++	++++
Set 4	No	++	++++	+++++
Set 5	Yes	++++	++	+++
Set 6	Yes	+++++	+	+
Set 7	Yes	++++	+++++	++
Set 8	Yes	++++	+	+

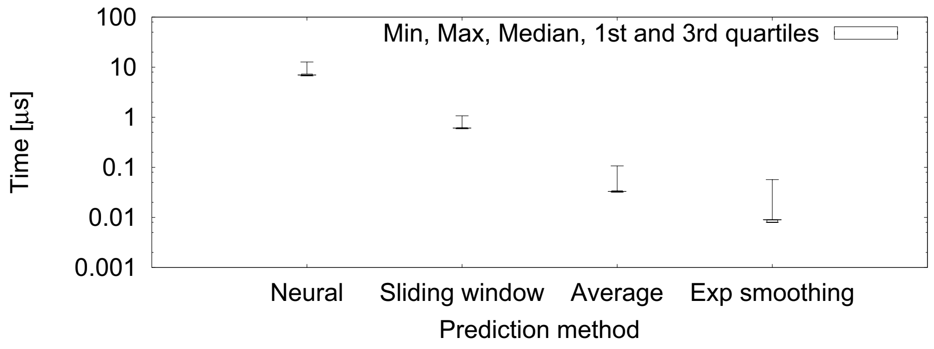
(see Figure 4). We use this simulator for generating realistic load patterns such as entity interaction hot-spots or simply large numbers of entities managed by one game server. The entities in the simulation are driven by several Artificial Intelligence (AI) profiles which determine their behaviour during a simulation: *aggressive* determines the entity to seek and interact with opponents; *team player* causes the entity to act in a group together with its teammates; *scout* leads the entity for discovering uncharted zones of the game world (not guaranteeing any interaction); and *camper* simulates a well-known tactic in FPS games to hide and wait for the opponent. The four profiles have been selected to match the four behavioural profiles most encountered in MMOGs [1]: the achiever, the explorer, the socialiser, and the killer, respectively. To also account for the mixed behaviour encountered in deployed MMOGs [1], each entity has its own preferred profile, but can change the profiles dynamically during the emulation. We further tried to get as close as possible to real games by importing maps from a very popular FPS game (Counter Strike 1.6 [5]).

We evaluated the prediction service using eight different data traces generated with our simulator for a duration of 17 hours with a sampling rate of two minutes (see Table 1). The first four data traces simulate different scenarios of a highly dynamic FPS game, while the other four are characteristic to different MMORPG sessions. We compared the error of the neural network prediction against other fast prediction methods such as moving average, last value, and exponential smoothing, which have been proven to be effective in large dynamic environments as the Grid [11]. Each prediction algorithm receives as input each trace data set, and outputs for each input set sample a prediction. For each prediction algorithm and trace data set combination, we define the *prediction error* as the ratio between the sum of all sample prediction errors and the sum of all N samples in the trace data set (expressed as a percentage): $PE = \frac{\sum_{i=1}^{N} \left| n_i^{real} - n_i^{pred} \right|}{\sum_{i=1}^{N} n_i^{real}} \cdot 100$, where n_i^{real} and n_i^{pred} denote the real, respectively predicted entity counts at time step i.

The results shown in Figure 5(a) shows that, apart from producing better or at least equal predictions, the important quality of our method is its ability to



(a) Prediction comparison



(b) Prediction time

Fig. 5. Neural network-based prediction results

adapt to various types of input signals, which the other methods fail to achieve. Even though they often produce good results, the drawback of the other methods is that it is not universally clear during a game play which of them should be applied as the real-time prediction method for the next time step. Moreover, as the dynamics of the game may change, for example during peak hours, the best prediction method may change too. Our neural network-based prediction successfully manages to adapt to all these types of signals and always delivers good prediction results. Furthermore, the best results were obtained for the more dynamic signals which best characterise the FPS games.

Figure 5(b) depicts the duration of one prediction on a common off-the-shelf desktop workstation with an Intel Core Duo E6700 (2.66 gigahertz) processor for all discussed prediction methods except last value which has no computational requirements. Although the neural network predictor is the slowest of them, it is nevertheless extremely fast with an average prediction duration of approximately 7 microseconds which makes it suitable for the severe real-time requirements of online games.

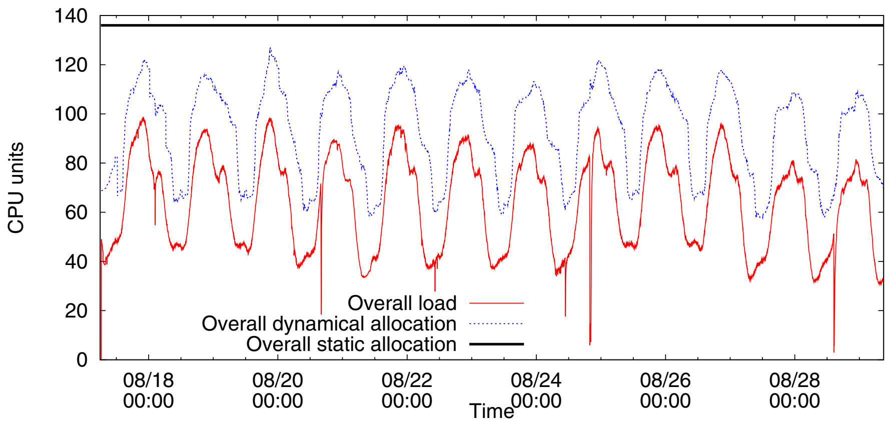


Fig. 6. Static versus dynamic resource allocation

7.2 Resource Allocation

In evaluating the efficiency of the resource allocation, we used traces collected from the official Web page of an MMORPG game called RuneScape [7]. RuneScape is not a traditional MMORPG, but consists of several minigames combining elements of RPG and FPS. Thus, various levels of player interactivity coexist in the same game and the game load cannot be trivially computed with the linear models employed in [12], but require more sophisticated methods like we presented in Section 5. The traces contain the number of players over time for each server group used by the RuneScape game operators[1]. For this work, we have analysed over six months of data until March 2008 with the metrics being evaluated every two minutes giving over 10 thousand samples for each simulation ensuring statistical soundness. We used in our experiments a minimum resource bulk of 0.25 and a time bulk of six hours (i.e. deallocation cannot be done earlier, as explained in Section 6).

To quantify the effectiveness of resource allocation, we measure the over-allocation as the percentage allocated from the total amount of resources necessary for the seamless execution of the MMOG that maintains the real-time QoS requirements. We define the total resource *over-allocation* $\Omega(t)$ at time instance t as the cumulated over-allocation of all machines participating in the game session, where M is the number of machines in the session, $\alpha_m(t)$ represents the allocated resource on machine m and $\lambda_m(t)$ represents the resource usage (the generated load) on machine m: $\Omega(t) = \frac{\sum_{m=1}^{M} \alpha_m(t)}{\sum_{m=1}^{M} \lambda_m(t)} \cdot 100$.

Figure 6 shows comparatively the static and dynamic resource allocation for the same workload. As expected, the dynamic resource allocation is better than the static over-provisioning strategy. The average over-allocation is drastically

[1] We could not use this traces for the load prediction validation since the zones on one server group are too large for an accurate prediction and contain no entity position information.

reduced from 250% in case of static over-provisioning to around 25% (mostly due to the six hour time bulk) for the dynamic allocation strategy.

8 Conclusions

We proposed a new prediction-based method for dynamic resource provisioning and scaling of real-time MMOGs in distributed Grid environments. Firstly, we developed a load prediction service that accurately estimates the future game world entity distribution from historical information using a fast and flexible neural network-based approach. Apart from the ability to adapt to a wide range of signals characteristic to different game genres, styles, and user loads, our method is also extremely fast which makes it suitable to applications with real-time requirements like online games. On top of it, we developed generic analytical game load models used to foresee future hot-spots that congest the game servers and make the overall environment fragmented and unplayable. Based on the load prediction information, a resource allocation service performs dynamic provisioning, proactive load balancing, and migration of entities that keep the game servers reasonably loaded to maintain the real-time QoS requirements. Using our allocation method, we demonstrated a 10-fold improvement in resource provisioning for a real-world MMORPG game.

References

1. Bartle, R.: Designing Virtual Worlds. New Riders Games (2003)
2. Inc. Blizzard Entertainment. World of warcraft, http://www.worldofwarcraft.com/
3. Cai, W., Xavier, P., Turner, S.J., Lee, B.S.: A scalable architecture for supporting interactive games on the internet. In: PADS 2002: Proceedings of the sixteenth workshop on Parallel and distributed simulation, pp. 60–67. IEEE Computer Society Press, Washington (2002)
4. Feng, W., Brandt, D., Saha, D.: A Long-Term Study of a Popular MMORPG. In: Proceedings of NetGames 2007, Netgames, pp. 19–24 (2007)
5. Inc. GameData. Counter strike, http://www.counter-strike.com
6. Glinka, F., Ploss, A., Müller-Iden, J., Gorlatch, S.: A real-time framework for developing scalable multiplayer online games. In: NetGames. ACM Press, New York (2007)
7. Ltd. Jagex. Runescape (November 2007), http://www.runescape.com
8. Müller-Iden, J., Gorlatch, S.: Rokkatan: scaling an RTS game design to the massively multiplayer realm. Computers in Entertainment 4(3), 11 (2006)
9. MMORPG.COM. Your headquarters for massive multiplayer online role-playing games, http://www.mmorpg.com/
10. White, W.M., Koch, C., Gehrke, J., Demers, A.J.: Database research opportunities in computer games. SIGMOD Record 36(3), 7–13 (2007)
11. Wolski, R.: Experiences with predicting resource performance on-line in computational grid settings. ACM SIGMETRICS Performance Evaluation Review 30(4), 41–49 (2003)
12. Ye, L., Cheng, M.: System-performance modeling for massively multiplayer online role-playing games. IBM Systems Journal 45(1) (2006)

2D Fast Poisson Solver for High-Performance Computing

Alexander Kalinkin[1], Yuri M. Laevsky[2], and Sergey Gololobov[1]

[1] Intel Corporation, Lavrentieva ave. 6/1, 630090 Novosibirsk, Russia
[2] Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Lavrentieva ave. 6
laev@labchem.sscc.ru

1 Introduction

Among elliptic boundary value problems, the class of problems with separable variables can be solved fast and directly. Elliptic problems with separable variables usually assume that the computational domains are simple e.g., rectangle or circle, and constant coefficients [1]. This kind of problems can serve to generate preconditioners in iterative procedures for more complex methods. They can also be used in low-accuracy models similar to the ones used in Numerical Weather Simulations. A straightforward implementation of solvers for such problems makes them considered as too simple to pay much attention to them. For instance, NETLIB* contains the codes of various solvers for problems with separable variables. We are not aware if such solvers are provided as separate solvers in other software packages except NETLIB Fishpack* and Intel® Math Kernel Library (Intel® MKL) together with Intel Cluster Poisson Solver Library (Intel® CPSL). The problems with the separable variables can be suboptimal in the sense that they require slightly more arithmetic operations (up to logarithmic factor) than the number of unknowns to compute the solution to the problem. This statement is true if, for example, the sizes of the discretized problems are powers of 2. Computational Mathematics suggests that we take into consideration not only arithmetic operations, but also the cost of memory operations as well. Modern computers can perform computations at a very high speed, while lacking the ability to deliver data to the computational units. Keeping in mind that a memory operation can easily be dozen to hundred times slower than an arithmetic one, a computationally optimal algorithm could compute the solution slower than memory optimal algorithm. The recent developments in processor industry resulted in multicore processors become standard processors not only for powerful clusters, but also for home computers and laptops. Therefore, the algorithm can also be non-optimal from the parallelization point of view. Optimality here can be understood in terms of the number of synchronization points and/or in terms of the amount of data that needs to be transferred between different cores/processors. In summary, the modern computational algorithm should focus on 3 key aspects, namely, parallelization, memory operations, and arithmetic costs. The purpose of this paper is to demonstrate on a simple 2D problem with separable variables that taking into account modern model the solution can be computed efficiently and fast. This would also help developers to compute solution to the problem with separable variables really negligible with respect to other computations. To complete our goal, we will evaluate software provided by Intel Corporation. In particular, we focus on the comparison (where possible) of NETLIB Fishpack* and Intel® CPSL provided at [2] as well as Intel MKL provided at [3]. This paper is organized as follows. First we are solving a 2D Helmholtz problem using single precision on a shared memory machine with two 4-Core Intel® Xeon® Processors E5462. Next we do the same but use double precision arithmetic. Lastly we use double precision on a distributed memory machine using MPI.

2 Problem Statement

We are going to use the following notation for boundaries of a rectangular domain $a_x < x < b_x, a_y < y < b_y$ on a Cartesian plane:

$$\begin{aligned} &bd_{a_x\psi} = x = a_x, a_{y\psi} \le y \le b_y, bd_{b_{x\psi}} = x = b_x, a_{y\psi} \le y \le b_y, \\ &bd_{a_y\psi} = y = a_y, a_{x\psi} \le x \le b_x, bd_{b_{y\psi}} = y = b_y, a_{x\psi} \le x \le b_x \rhd . \end{aligned} \tag{1}$$

The wildcard "*" may stand for any of the symbols ax, bx, ay, by, so that bd_* denotes any of the above boundaries. The 2D Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + qu = f(x,y), q = const \ge 0, \tag{2}$$

in a rectangle, that is, a rectangular domain $a_x \le x \le b_x, a_y \le y \le b_y$, with one of the following boundary conditions on each boundary bd_*:

The Dirichlet boundary condition: $u(x,y) = G(x,y)$
The Neumann boundary condition: $\frac{\partial u}{\partial n} = g(x,y)$

$n = -x$ on bd_{a_x}, $n = x$ on bd_{b_x} and $n = -y$ on bd_{a_y}, $n = y$ on bd_{b_y}

We can see that the Poisson problem can be obtained from the Helmholtz problem by setting the Helmholtz coefficient q to zero. The Laplace problem can be obtained by setting the right-hand side f to zero in addition to Helmholtz coefficient. To find an approximate solution for 2D problems, a uniform mesh is built in the rectangular domain:

$$\begin{aligned} &x_i = a_x + ih_x, y_j = y(j) \\ &i = 0, \ldots, n_x, h_x = \frac{b_x - a_x}{n_x}, \\ &a_y = y(0), < \ldots < y(j) < \ldots < y(n_j) = b_j \end{aligned} \tag{3}$$

It is possible to use the standard five-point finite difference approximation on this mesh to compute the approximation to the solution. We assume that the values of the approximate solution are computed in the mesh points (x_i, y_j), provided the values of the right-hand side $f(x, y)$ at these points are given and the values of the appropriate boundary functions $G(x, y)$ and/or $g(x, y)$ in the mesh points laying on the boundary of the rectangular domain are known. We conducted our measurements on two 4-Core Intel® Xeon® Processors E5462 (12M Cache, 2.80 GHz, 1600 MHz FSB) equipped with 16GB of RAM. All test cases were compiled with Intel® C/C++ and Fortran compilers (version 10.1.018). We used the following set of options -xT -O3 -ipo -no-prec-div static recommended for the best performance. We ran each piece of code 4 times in a loop and then selected the best time out of the four collected. Time measurements were completed using the dsecnd routine from Intel® MKL 10.1 Update 1^+. MPI measurements were completed with MVAPICH* 1.0 and the corresponding MPI _ Wtime routine (single run). We also used the Poisson Library from Intel® MKL 10.1 Update 1, the Intel® Cluster Poisson Solver Library from whatif.intel.com (release 1.0.0.003) and NETLIB Fishpack* library built with Intel® Fortran compiler mentioned above. Discrete Fourier Transform (DFT) computations are highly dependent on the dimension. For powers of 2, the DFT requires the least possible number of operations, while for the primes the number of operations reaches its maximum value. We consider only dimensions that are powers of 2 as the difficult test case with a high data movement to operations ratio.

3 Single Precision SMP Results

We first look at single precision computations that are of value for Numerical Weather Simulation problems and consider the hwscrt routine from NETLIB Fishpack* and the Poisson Library (PL) from Intel® MKL. The hwscrt routine is able to compute the solution to the 2D Helmholtz problem in a Cartesian coordinate system in a single step. PL does computations in four steps by consecutive calls to the s_init_helmholtz_2d, s_commit_helmholtz_2d, s_helmholtz_2d, and free_helmholtz_2d routines. For fairness, we measure the total time spent in computations for both software libraries. We consider the homogeneous Dirichlet problem with an exact solution on the rectangular domain $0 < x < 1, 0 < y < 1$ as our test case. The first figure shows the ratio of computational time spent in the hwscrt routine and computational time spent in four PL routines for different regular mesh sizes starting from 4x4 and ending with 8192x8192 and with different numbers of OMP threads. We dont refine the mesh further as the accuracy of computations degrades for a larger number of mesh points. When the ratio curve is above 1, PL Helmholtz 2D solver is faster than NETLIB Fishpack*. Figure 2 shows scalability of Intel® MKL PL routines in 2D Cartesian case. As the hwscrt routine from NETLIB Fishpack* is not threaded, there is very little difference in run time for different numbers of OMP threads in this routine. It is worth mentioning that the routine can be threaded, however that would require considerable additional work. From Figure 1, it can be seen that even on a single thread, the PL routines can be up to 5 times faster than the hwscrt routine from NETLIB Fishpack*. When threaded, the advantage grows up to 25 times on 8 threads. It is also clear that PL has heavier interface that results in essentially slower performance for small problems (sizes up to 128). However, PL can regain some performance in the case when the solution of problems different in right-hand side only as it can do pre- (init and commit) and post-computational (free) step only once unlike the hwscrt routine.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations.

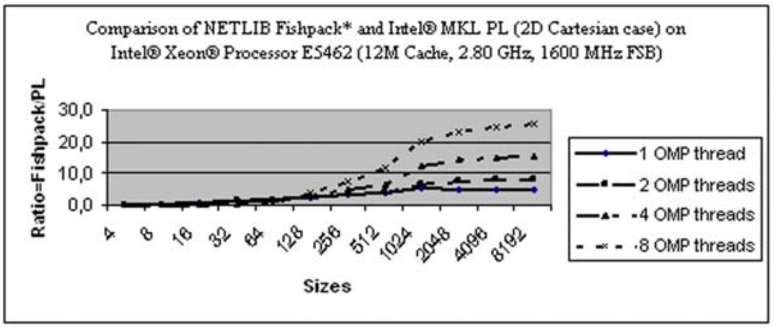


Fig. 1. Comparison of NETLIB Fishpack* and Intel® MKL (2D Cartesian case) (single precision)

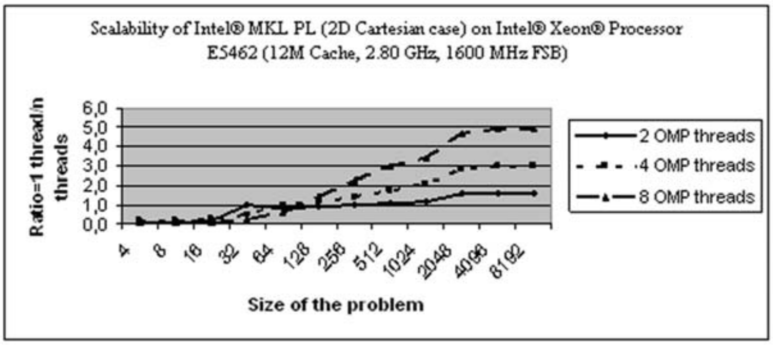


Fig. 2. Scalability of Intel® MKL PL (2D Cartesian case) (single precision)

Table 1. Computational time and error on 1 thread (single precision)

size NxN	Pl time(s)	Fishpack* time(s)	Max norm of error for PL	Max norm of error for Fishpack*
32	3,78E-05	5,52E-05	7,34E-04	7,34E-04
128	4,44E-04	1,21E-03	1,34E-04	5,45E-05
512	6,69E-03	2,57E-02	2,50E-04	4,04E-01
2048	1,61E-01	7,97E-01	5,98E-03	9,45E-01
8192	2,96E-00	1,52E+01	1,59E-01	9,99E-01

Table 2. Computational time and error on 8 thread (single precision)

size NxN	Pl time(s)	Fishpack* time(s)	Max norm of error for PL	Max norm of error for Fishpack*
32	1,52E-04	5,55E-05	7,34E-04	7,34E-04
128	3,17E-04	1,21E-03	1,34E-04	5,45E-05
512	2,20E-03	2,58E-02	2,50E-04	4,04E-01
2048	3,47E-02	8,00E-01	5,98E-03	9,45E-01
8192	6,00E-01	1,52E+01	1,59E-01	9,99E-01

Again, it can be clearly seen that multiple threads worse of turning on only for larger sizes of problems (starting from 128). However, later the scalability of PL gets really good (up to 5 times speed-up for 8 threads vs. 1 thread). It should be kept in mind that for very small times of order (-6)(-4) variations of measurements is relatively strong. We did not take any special steps to stabilize our measurements as for such small computational times it is not so important if the computational time is 1.0E-06 or 5.0E-06. The data related to performance run on 1 thread are contained in Table 1:

The data related to performance run on 8 threads are contained in Table 2:

From the Tables above, we can see the substantial drop in accuracy in the hwscrt routine at size 512. The accuracy of computations becomes inacceptable starting at size 2048 (of order of magnitude of the seek solution).

We note that PL has large interface overhead for small size problems as there is no difference between computational times for the sizes varying from 32 up to and including 128. We think that small size problems are not of great interest for HPC area, so the lower performance of PL in this range should not be considered as a problem.

We can also conclude that single precision computations do not actually require a cluster because of a small size of the problem (up to 64Mx4bytes per element 256MB of memory). However, cluster software may benefit from having Helmholtz solver for distributed memory machines to avoid excessive communications.

4 Double Precision SMP Results

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations.

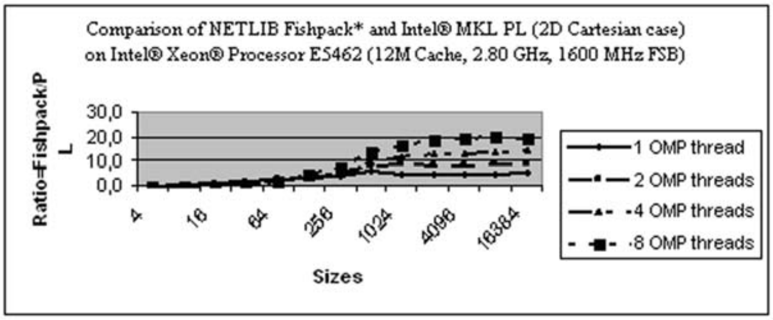


Fig. 3. Comparison of NETLIB Fishpack* and Intel® MKL (2D Cartesian case) (double precision)

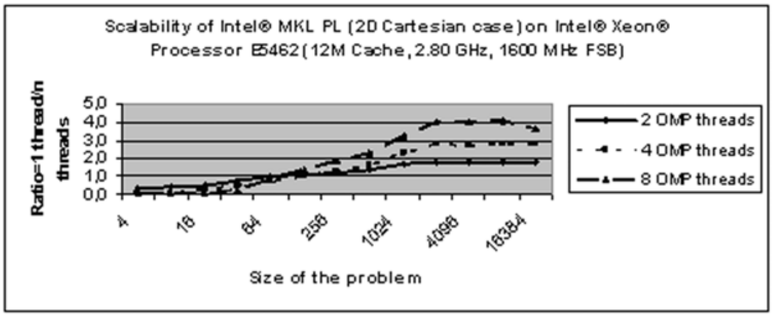


Fig. 4. Scalability of Intel® MKL PL (2D Cartesian case) (double precision)

We next look at double precision computations that are of value for preconditioning of elliptic problems with slightly varying coefficients and we consider again the hwscrt routine from NETLIB Fishpack* and the Poisson Library from Intel® MKL. PL does computations in four steps by consecutive calls to the double precision routines d_init_helmholtz_2d, d_commit_helmholtz_2d, d_helmholtz_2d, and free_helmholtz_2d. We measured the total time spent in computations for both software libraries.

We consider the same homogeneous Dirichlet problem with the same exact solution on the same rectangular domain $0 < x < 1, 0 < y < 1$ as our test case. Figure 4 shows the ratio of computational time spent in the hwscrt routine and computational time spent in four PL routines for different regular mesh sizes starting from 4x4 and ending with 16384x16384 and with different numbers of OMP threads. We can refine the mesh further as the accuracy of computations is good enough at least for PL, but the time for computations using the hwscrt routine would be too big. When the ratio curve is above 1, PL Helmholtz 2D solver is faster than NETLIB Fishpack*.

Figure 4 shows scalability of PL routines in 2D Cartesian case.

From the Figures above, we can come to the similar conclusions as for single precision computations. We can also conclude that double precision computations could be of interest for the cluster computations because the problem size can be huge enough for a single machine to handle it. And this is the topic for our next consideration.

5 Double Precision MPI Results

At last, we look at double precision computations for distributed memory machine. We cannot consider the hwscrt routine from NETLIB Fishpack* anymore as it is not designed for such kind of computations. Therefore, Intel® CPSL software will be evaluated only. Intel® CPSL does computations in four steps by consecutive calls to the double precision routines dmv0_init_helmholtz_2d, dmv0_commit_helmholtz_2d, dmv0_helmholtz_2d, and dmv0_free_helmholtz_2d. We measured the computational time only (the time spent in the dmv0_helmholtz_2d routine).

We consider the same homogeneous Dirichlet problem with the exact solution on the rectangular domain $u(x, y) = sin(\pi x)sin(\pi y)$,$0 < x < 1, 0 < y < 1$ as our test case.

Following the instruction, we properly distributed the array f that initially contains the right-hand side of the problem. This array is represented as a rectangle $(1 \ldots nx + 1, 1 \ldots ny + 1)$ containing the values of the corresponding function. It is distributed among MPI processes so that the MPI process with rank p $(p = 0, \ldots, nproc - 1)$ contains the sub-array $f(1 \ldots nx + 1, n_p \ldots n_{p+1} - 1)$, where

$$\begin{aligned} n_1 = 1,\, n_{p+1} &= n_p + [\tfrac{nx+1}{nproc}] + 1, \; ifp < \tfrac{nx+1}{nproc}, \\ n_{p+1} &= n_p + [\tfrac{nx+1}{nproc}], \qquad ifp \geq \tfrac{nx+1}{nproc}. \end{aligned}$$

Here $[\frac{u}{v}]$ is the integer part of $\frac{u}{v}$, $\{\frac{u}{v}\}$ is the remainder $u - [\frac{u}{v}] * v$, and $nproc$ is the number of MPI processes.

It is obvious that for the dimensions that are powers of 2, the scalability of the software on such small problems cannot be good as the communications will be dominating over the computations. Therefore, it is better to use the Poisson Library from Intel MKL for solving the problems of the limited size. In order

Table 3. Summarized results

OMP threads\MPI processes	16	32	64	128
1	1,58E+02	5,57E+01	2,70E+01	1,24E+01
2	1,06E+02	5,14E+01	1,76E+01	8,72E+00
4	7,72E+01	2,76E+01	1,32E+01	6,65E+00
8	6,61E+01	2,50E+01	1,28E+01	6,38E+00

to get the real value of the Intel® CPSL, we took the problem of size 10^5x10^5, that is, the problem with 10^{10} unknowns and ran it. The results are summarized in Table 3:

We can see that OMP scalability is poor. However, MPI scalability is perfect and presented on Figure 5 for single thread per MPI process. Super linear speed-up is explained by the effect from the memory swap. The more processes are involved in computations the more data can fit into the computer memory. Therefore, the computations are performed faster.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations.

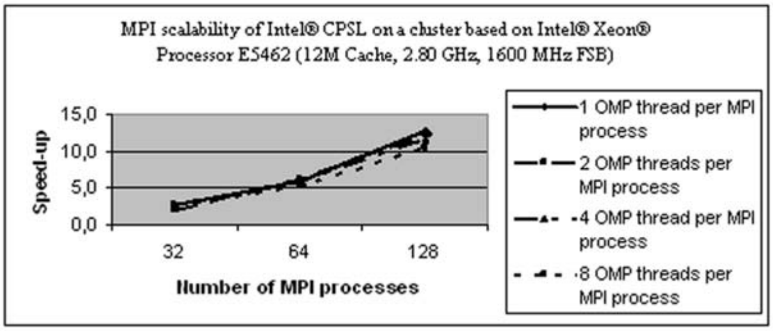


Fig. 5. Scalability of Intel® MKL PL (2D Cartesian case) (double precision)

6 Conclusions

We have investigated the performance of software available for solving 2D Helmholtz problems with separable variables on rectangular domains. We found that the straightforward implementation of the solver from NETLIB Fishpack* shows good performance for small size problems (up to 128 mesh points in one dimension). The more sophisticated implementation of Intel® MKL PL shows better performance and scalability on larger problem sizes. Performance gains are from 5 up to 25 times for some dimensions. It also provides better accuracy for large problem sizes.

We found that Intel CPSL provides additional benefits for distributed memory machines. While hybrid (MPI+OMP) performance is not improving with the growing number of threads, the pure distributed memory (MPI only) performance demonstrates extremely good scalability results.

References

1. Samarskii, A.A., Nikolaev, E.S.: Methods of Solution of Grid Problems. Nauka, Moscow (1978) (in Russian)
2. Intel Cluster Poisson Solver Library, http://software.intel.com/en-us/articles/intel-cluster-poisson-solver-library
3. Intel Math Kernel Library, http://www.intel.com/software/products/mkl

Solution of Large-Scale Problems of Global Optimization on the Basis of Parallel Algorithms and Cluster Implementation of Computing Processes

Vladimir Koshur, Dmitriy Kuzmin, Aleksandr Legalov, and Kirill Pushkaryov

Siberian Federal University, Institute of Space and Information Technology,
Kirenskiy St., 26, Krasnoyarsk, 660074, Russia
VKoshur@sfu-kras.ru

Abstract. The parallel hybrid inverse neural network coordinate approximations algorithm (PHINNCA) for solution of large-scale global optimization problems is proposed in this work. The algorithm maps a trial value of an objective function into values of objective function arguments. It decreases a trial value step by step to find a global minimum. Dual generalized regression neural networks are used to perform the mapping. The algorithm is intended for cluster systems. A search is carried out concurrently. When there are multiple processes, they share the information about their progress and apply a simulated annealing procedure to it.

Keywords: optimization, global optimization, large-scale problems solution, cluster, neural networks.

1 Introduction

Global optimization is important in both theory and praxis. It involves more complex algorithms than local optimization, but results are of great value. Sometimes locating of local extremes is sufficient. At the same time, there are tasks where global optimization is indispensable, for example, computer-assisted proofs, safety verification, and molecule conformation search [1]. Nowadays there are many global optimization methods, such as evolutionary algorithms [2], the simulated annealing [3], the Monte Carlo method [1], the coordinates averaging method [4].

Global optimization of unsmooth functions with multiple extremes over subsets of N-dimensional space (N = 10^2–10^6) is a challenging problem of computer science. Usually practical objective functions are complex, and their evaluation requires partial simulation of a system that is being optimized. An aerodynamic shape optimization (7–10 variables in the very simplified case [5]), a protein folding problem (3M variables, where M is a number of molecules), a nonlinear least squares problem (dimensionality is proportional to a training set size in training neural networks) are computation-intensive. Effective parallel algorithms and procedures are necessary to solve such problems. Nowadays clusters are widely employed in high performance computing, as they are powerful, scalable and relatively inexpensive [6]. 410 of 500 world supercomputers were clusters as of November 2008 [7]. So, it seems particularly important to develop applications for such architectures.

In this paper we present the parallel hybrid inverse neural network coordinate approximations algorithm (PHINNCA), a new global optimization algorithm, which we consider promising. In the work [8] it was shown that inverse dependencies approximation may be used to perform a global optimization. The new algorithm is based on the inverse neural network coordinate approximations algorithm (INNCA) [9], which is being developed by the authors, and the simulated annealing technique [3]. The algorithm is well suited for cluster implementation.

2 Synopsis of the PHINNCA

The hybrid algorithm combines the INNCA with the simulated annealing technique. A search is carried out in parallel (N processes are shown in Fig. 1). Multiple processes perform inverse coordinate approximations concurrently. The processes continually exchange the best results that they achieved. A process adopts the results that are worse than its own with certain probability (4) and thus it does an annealing.

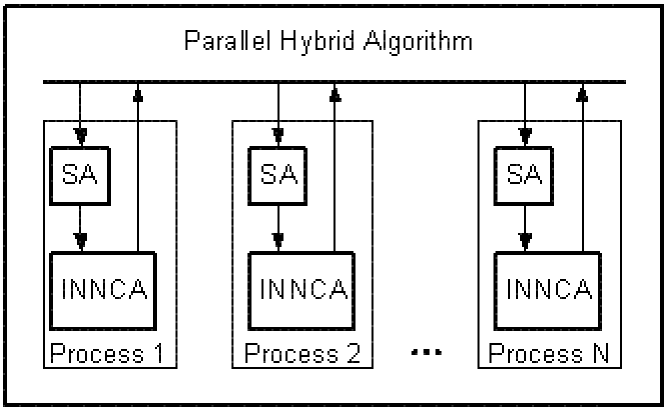


Fig. 1. Structure of the parallel program. Each process performs inverse neural network coordinate approximations (INNCA), shares its best results with other processes, and does a simulated annealing (SA) on the base of results received from others.

The program is suitable for cluster execution. The authors implemented it using the "Parallel Computing Toolbox" [10] of the MATLAB system.

3 Description of the PHINNCA

Suppose we have the bounded function $f(\vec{\mathbf{x}}) : \Omega \to R$, where $\vec{\mathbf{x}} = (x_1, x_2, ..., x_n) \in \Omega \subset R^n$. Further assume that the function is continuous almost everywhere in the bounded area $\Omega = \{(x_1, x_2, ..., x_n) : x_i \in [a_i, b_i], i = \overline{1,n}\}$. We must find the point of global minimum $\vec{\mathbf{x}}_{\min}$

$$f_{\min} = f(\vec{\mathbf{x}}_{\min}) = \min_{\vec{\mathbf{x}} \in \Omega} f(\vec{\mathbf{x}}). \tag{1}$$

The algorithm for approximate calculation of $\vec{\mathbf{x}}_{\min}$ is iterative. Define the set of candidate points at the k-th iteration $X^{[k]}$ as

$$X^{[0]} = B^{[0]}; X^{[k]} = X^{[k-1]} \cup B^{[k]} \cup H^{[k]} \cup P^{[k]}. \tag{2}$$

The points from $B^{[k]}$ are generated by blind search and may be random or form a uniform mesh. They are intended to reveal the behaviour of the objective function in the search space. The information accumulated about the behaviour contributes to the heuristic search. The heuristic search generates points for $H^{[k]}$ set. Multiple search processes running in parallel exchange information about the best candidate solutions discovered so far. That information ends up in $P^{[k]}$ set.

The set of prediction functions Φ_i is a core of the heuristic search. For each coordinate we have

$$x_i = \Phi_i(f_\varphi), (i = 1,2,...,n), f_\varphi \in R, \vec{\mathbf{x}} \in \Omega. \tag{3}$$

The functions map values of the objective function to values of its arguments. Taking (3) into account, we require the prediction functions to be such that $\vec{\mathbf{x}} \xrightarrow[f_\varphi \to f_{\min}]{} \vec{\mathbf{x}}_{\min}$. With that functions, we decrease f_φ, a trial value of the objective function, step by step to find a global minimum.

The authors invented DGRNN (Dual Generalized Regression Neural Networks), a modification of GRNN (Generalized Regression Neural Networks) [11], to approximate the prediction functions (3). In contrast with GRNN, DGRNN has an additional input, a focal point. The point must be a center of area of interest. A DGRNN operates in a neighbourhood of a focal point while an influence of remote items of a training set is suppressed. Fig. 2 demonstrates the behaviour of DGRNN and GRNN prediction functions for the function $f_1(x) = (x-1)^2 \cdot (x+1)^2 + 1$.

Recall that the search processes exchange information about their progress, and the best points received at the k-th iteration from the other processes accumulate in $P^{[k]}$. So, we find the best of them, $\vec{\mathbf{p}}^{[k]} = \arg\min_{\vec{\mathbf{x}} \in P^{[k]}}(f(\vec{\mathbf{x}}))$.

A procedure analogous to simulated annealing [2] is employed by a process to decide whether it should accept the best of the points received ($\vec{\mathbf{p}}^{[k]}$) or proceed with a locally generated value ($\vec{\mathbf{x}}^{[k]}$). The point is accepted with a probability $P_a(f(\vec{\mathbf{p}}^{[k]}) - f(\vec{\mathbf{x}}^{[k]}))$,

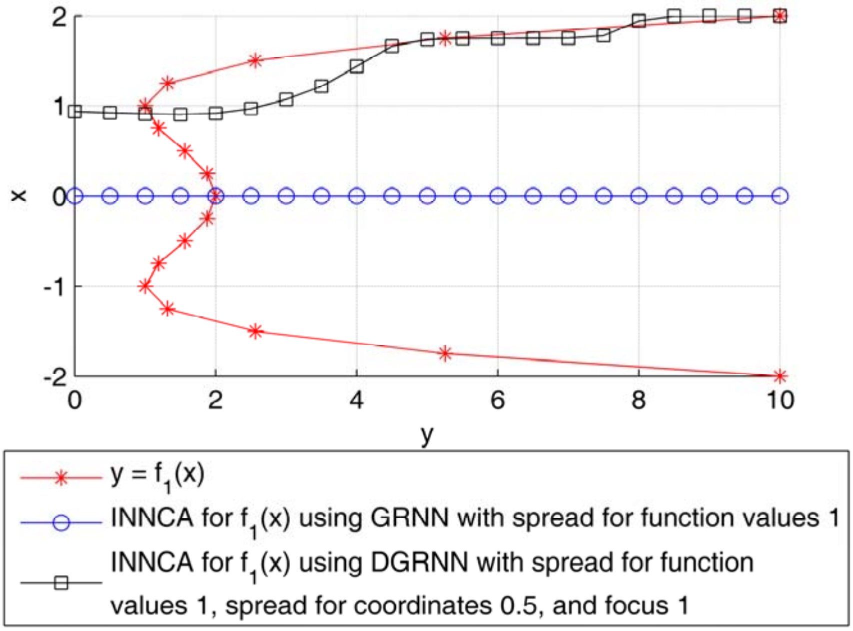


Fig. 2. Inverse neural network coordinate approximations for the function $f_1(x)$

$$P_a(x) = \begin{cases} 1 & \text{if } x < 0, \\ e^{\frac{-x}{T_0/k}} & \text{else,} \end{cases} \tag{4}$$

where T_0 is an initial annealing temperature.

A process stops when a progress speed falls below the threshold, or the limit of iterations is reached, or value of the objective function is satisfactorily low.

Finally, when the search stops at the S-th iteration, the final answer is $\hat{\vec{\mathbf{x}}}_{\min} = \arg\min_{\vec{\mathbf{x}} \in X^{[S]}} f(\vec{\mathbf{x}})$.

4 Conclusion

Global optimization problems are computation-intensive. So, global optimization applications are good candidates for parallel implementation.

In this article we inquired into application of the parallel hybrid inverse neural network coordinate approximations algorithm to global optimization. The algorithm combines the INNCA algorithm with simulated annealing and is well suited for running on clusters. It uses the DGRNN, a modified GRNN, to approximate inverse relations. The authors regard the algorithm as a promising routine for solution of large-scale problems of global optimization.

References

1. Neumaier, A.: Complete Search in Continuous Global Optimization and Constraint Satisfaction, http://www.mat.univie.ac.at/~neum/glopt/mss/Neu04.pdf
2. Wang, H., Ersoy, O.: Novel Evolutionary Global Optimization Algorithms and Their Applications, http://docs.lib.purdue.edu/ecetr/340/
3. Russel, S., Norvig, P.: Artifical Intelligence: A Modern Approach, 2nd edn. Williams Publishing House, Moscow (2006) (in Russian)
4. Ruban, A.I.: Global Optimization by Averaging of Coordinates (in Russian). IPC KGTU, Krasnoyarsk (2004)
5. Mengistu, T., Ghaly, W.: Global Optimization Methods for the Aerodynamic Shape Design of Transonic Cascades, http://www.mat.univie.ac.at/~neum/glopt/mss/MenG03.pdf
6. Voevodin, V.V., Voevodin, Vl.V.: Parallel Computing (in Russian). BHV-Petersburg, Saint Petersburg (2004)
7. Architecture share for 11/2008 | TOP500 Supercomputing Sites, http://www.top500.org/stats/list/32/archtype
8. Koshur, V.D.: Adaptive Algorithm of Global Optimization Based on Weighted Averaging of Coordinates and Fuzzy Neural Networks (in Russian). Electronic peer-reviewed journal "Neuroinformatika" 1(2), 106–124 (2006), http://www.niisi.ru/iont/ni/Journal/N2/Koshur.pdf
9. Koshur, V.D., Pushkaryov, K.V.: Global Optimization Based on Inverse Relations and Generalized Regression Neural Networks (in Russian). In: X-th All Russia Scientific and Technical Conference "Neuroinformatika 2008". MIFI, Moscow (2008)
10. MathWorks: Parallel Computing Toolbox: Programming Overview: Product Introduction: Overview, http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/brkl0o6.html
11. Medvedev, V.S., Potyomkin, V.G.: Neural Networks. MATLAB 6 (in Russian). Dialog-MIFI, Moscow (2002)

DEEP - Differential Evolution Entirely Parallel Method for Gene Regulatory Networks

Konstantin Kozlov[1,⋆] and Alexander Samsonov[2]

[1] Dept. of computational biology, State Polytechnical University, St.Petersburg, 195251 Russia
kozlov@spbcas.ru

[2] A.F. Ioffe Physico-technical Institute of the Russian Academy of Sciences, St.Petersburg, 194021, Russia
samsonov@math.ioffe.ru

Abstract. DEEP - Differential Evolution Entirely Parallel method is applied to the biological data fitting problem. We introduce a new migration scheme, in which the best member of the branch substitutes the oldest member of the next branch, that provides a high speed of the algorithm convergence. We analyze the performance and efficiency of the developed algorithm on a test problem of finding the regulatory interactions within the network of gap genes that control the development of early Drosophila embryo. The parameters of a set of nonlinear differential equations are determined by minimizing the total error between the model behavior and experimental observations. The age of the individuum is defined by the number of iterations this individuum survived without changes. We used a ring topology for the network of computational nodes. The computer codes are available upon request.

Keywords: differential evolution, optimization, regulatory gene networks.

1 Introduction

Differential evolution is an effective method for the minimization of various and complex quality functionals. Its power is based on the fact that under appropriate conditions it will attain the global extremum of the functional; its weakness is in high computational demand and dependence on control variables, that provides a motivation to parallelize DE. Previous work in this area has produced a number of methods that perform well in certain particular problems, but not in general applications.

Gene regulatory networks (GRNs) are the important set of models that has been developed for mathematical treatment and analyzing the developmental processes in biological objects. GRN represents the activation or repression of transcription of the gene product by other genes. The spatio-temporal dynamics of gene expression is described in the context of this study by the system of

⋆ Corresponding author.

highly nonlinear differential equations. Their parameters are to be found as a solution to the inverse problem of fitting the experimental gene expression data to computed model output.

We introduce a new migration scheme for the Differential Evolution Entirely Parallel (DEEP) method, that provides a high speed of the algorithm convergence. We present numerical results on optimization using the developed algorithm for the test problem of finding parameters in a network of two genes and the analysis of the dependency of the accuracy of the final result on the period of communication between branches. We show how changes in the quality of the answer computed in parallel can be compensated for by constructing a function relating the quality of the answer to the number of iterations required to obtain it in serial computations.

2 Methods and Algorithms

2.1 Differential Evolution Entirely Parallel Method

DE is a stochastic iterative optimization technique proposed by Storn and Price [1], that starts from the set of the randomly generated parameter vectors q_i, $i = 1, ..., NP.$. The set is called population, and the vectors are called individuals. The population on each iteration is referred to as a generation. The size of population NP is fixed. The first trial vector is calculated by:

$$v = q_{r1} + S(q_{r2} - q_{r3}) \tag{1}$$

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 members of population. The second trial vector is calculated using "trigonometric mutation rule" [2].

$$\begin{aligned} z = {} & \frac{q_{r1} + q_{r2} + q_{r3}}{3} + (s_2 - s_1)(q_{r1} - q_{r2}) \\ & + (s_3 - s_2)(q_{r2} - q_{r3}) + (s_1 - s_3)(q_{r3} - q_{r1}) \end{aligned} \tag{2}$$

where $s_i = |F(q_{ri})|/s^*$, $i = 1, 2, 3$, $s^* = |F(q_{r1})| + |F(q_{r2})| + |F(q_{r3})|$. The third trial vector is defined as follows:

$$w_j = \begin{cases} v_j, & j = \langle n \rangle_I, \langle n+1 \rangle_I, ..., \langle n+L-1 \rangle_I \\ z_j & j < \langle n \rangle_I \; OR \; j > \langle n+L-1 \rangle_I \end{cases} \tag{3}$$

where n is a randomly chosen index, $\langle x \rangle_y$ is the reminder of division x by y and L is determined by $Pr(L = a) = (p)^a$ where p is the probability of crossover. The new individuum replaces its parent if the value of the quality functional for its set of parameters is less than that for the latter one.

The original algorithm was highly dependent on internal parameters as reported by other authors, see, for example [3]. An efficient adaptive scheme for selection of internal parameters S and p based on the control of the population diversity was developed in [4] where a new control parameter γ was introduced.

Being an evolutionary algorithm, DE can be easily parallelized due to the fact that each member of the population is evaluated individually. The whole population is divided into subpopulations that are sometimes called islands or branches, one per each computational node. The individual members of branches are then allowed to migrate, i.e. move, from one branch to another according to predefined topology [5]. The number of iterations between migrations is called communication period Π.

We have developed a new migration scheme for the Parallel Differential Evolution in which the best member of one branch is used to substitute the oldest member of the target branch. The age of the individuum in our approach is defined by the number of iterations this individuum survived without changes. The computational nodes are organized in a ring and individuals migrate from node k to node $k+1$ if it exists and from the last one to the first one. Calculations are stopped in case that the functional F decreases less than a predefined value ρ during M steps.

The effect of parallelization is measured with respect to the number of the evaluations of functional Q as the most time consuming operation in the algorithm [6]. The parallel DE is considered as the converged one if one of the branches has converged. Then Q_p equals to the number of functional evaluations of the converged branch. For different number of nodes N speedup is defined as $A(N) = Q_s(\hat{F}_p(N))/\hat{Q}_p(N) * 100\%$ and parallel efficiency: $E(N) = A(N)/N * 100\%$, where hat sign ($\hat{}$) denotes the average over a set of runs, subscripts s and p denote serial and parallel runs respectively, and F denotes the final value of the functional.

2.2 Gene Regulatory Network Model

Segmentation genes in the fruit fly *Drosophila* control the development of segments, which are repeating units forming the body of the fly [7]. Immediately following the deposition of a *Drosophila* egg, a rapid series of 13 almost synchronous nuclear divisions take place, without the formation of cells. The period between two subsequent nuclear divisions is called cleavage cycle.

The expression of segmentation genes is to a very good level of approximation a function only of distance along the anterior-posterior (A-P) axis (the long axis of the embryo quasi ellipsoid). This allows to use models with only one-dimensional array of nuclei along the A-P axis. Let us denote as $M(n)$ the number of nuclei under consideration in cleavage cycle n. This number varies with n as $M(n) = 2M(n-1)$.

Denoting the concentration of the ath gene product (protein) in a nucleus i at time t by $v_i^a(t)$, we write a set of ordinary differential equations for N zygotic genes as:

$$\begin{aligned}\frac{dv_i^a(t)}{dt} &= R^a g\left(\sum_{b=1}^{N} T^{ab} v_i^b(t) + m_i^a\right) - \lambda^a v_i^a(t) + \\ &+ D^a(n)\left[(v_{i-1}^a(t) - v_i^a(t)) + (v_{i+1}^a(t) - v_i^a(t))\right],\end{aligned} \tag{4}$$

where $a = 1, ..., N$; $i = 1, ..., M(n)$.

The first term on the right hand side of (4) describes the regulated rate of synthesis of protein from the ath gene. The function $g(\cdot)$ is to be a monotonic sigmoid ranging from zero to one, and we use the following form $g(y) = (1/2)\left(1 + y/\sqrt{y^2+1}\right)$. The regulation of gene a by gene b is characterized by the regulatory matrix element T^{ab}. The term m_i^a describes the aggregate regulatory effect of various maternal transcription factors on gene a in nucleus i, which is constant in time in most cases. The maximum rate of synthesis for protein a is given by the function R^a. The second term on the right hand side of equation (4) describes the degradation of ath protein, which is modeled as first order decay with rate λ^a. Diffusion of protein from nucleus i to two adjacent nuclei is described in the third term. The equations (4) are augmented with initial conditions, whose values depend on the precise biological situation being modeled.

The model was successfully used in [8] to describe formation of stripes by the pair-rule gene *even-skipped* as the result of regulation from gap and maternal genes. In [9,11], the pattern formation in the gap gene system was studied basing on the model. The data on gene expression in fruit fly *Drosophila* is available in *FlyEx* database [10].

3 Results

To study the convergence of the method in a lab conditions we produced the artificial gene expression data for the network of two genes in eight nuclei by integration the model equations, using the set of parameters that represents already known solution. We took the model output for 9 time moments to calculate the functional value. The parameters associated with one gene are fixed, so seven are sought by the optimizer. We used $\kappa = \max\limits_i \frac{|q^{true} - q^{opt}|}{|q^{true}|} * 100\%$ to measure the accuracy of the obtained approximation of parameter set q^{opt} in respect to the known solution q^{true}.

We have neglected the communication costs in our performance analysis made for the sample runs because in real applications the time of evaluating the quality functional is much larger than that for the communication needed for information exchange, which makes communication time indeed negligible.

3.1 Serial Convergence Curve

The serial algorithm was implemented in ANSI C programming language and run on Dell PowerEdge 2800 with 2 Xeons 2.4 GHz.

Due to an absence of an analytical model for this algorithm the optimal choice of communication period Π is an empirical process up to date. The effect of parallelization is essentially eliminated when Π is large. In the case of small Π, the divergence of the population will decrease too rapidly resulting in severe loss of quality of the results. We show that a suitable choice of Π can lead to very high efficiency.

In order to compensate any changes in the quality of the result because of parallelization, it is desirable to know the expected number of serial iterations corresponding to a particular value of a quality functional. Then the speedup can be calculated by dividing the number of expected serial iterations at the final value of a functional obtained in parallel by the average number of parallel iterations required.

In the problem of finding the parameters of gene regulatory networks the final value of the quality functional is affected by the number of algorithmic parameters and hence does not correspond to a unique number of iterations. We characterized the function $Q_s(\hat{F}_p(N))$ that gives the number of functional evaluations that the serial algorithm needs to obtain the same value of the quality functional as in parallel by an extensive series of numerical runs varying:

- quality criterion threshold ρ: 1e-2, 1e-3, 1e-4, 1e-5;
- quality criterion number of steps M: 50, 75, 100, 150;
- adaptive scheme control parameter γ: 0.90, 1.10, 1.20, 1.30;
- number of individuals in population NP: 70, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260, 270, 280;
- communication period C: 1, 2, 3, 4, 5, 7, 10, 15, 20, 25, 30, 35.

In our experiments we used communication in serial runs as it increases the convergence speed. For each combination of parameters we made 100 runs that equals the total number of $1,612,800$ runs. Results are plotted with points in Figure 1.

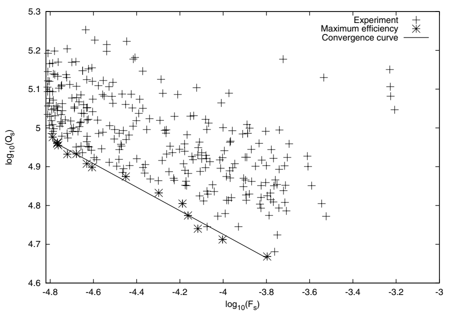


Fig. 1. Convergence curve. Each point represents one combination of parameters. $log_{10}(\hat{Q}_s)$ vs $log_{10}(\hat{F}_s)$.

The lower left-hand side of the graph shows a region where DE is most efficient, i.e. contains the desired characteristic relationship between the functional value and serial iterations for efficient evolution. Thus, the serial performance curve is constructed by fitting the points in that region to a power law. The data is well fitted by the equation (Figure 1):

$$log_{10}(\hat{Q}_s) = -0.3039 * log_{10}(\hat{F}_s) + 3.5099 \tag{5}$$

3.2 Parallel Performance

The parallel algorithm was implemented in ANSI C programming language and MPI was used for parallelization. Runs were performed with different combinations of parameters on the cluster of The Ioffe Physical-Technical Institute equipped with 24 Pentium III Xeons working at 2.0 GHz, 14 AMD AthlonMP 2400+ and 160 IBM PowerPC 2200 processors. Table 1 shows best results with respect to κ.

Table 1. Best optimization results for test problem in respect to solution quality κ for different number of nodes N. The following parameters are given for each case: stopping criterion parameters M and ρ, control parameter γ, communication period C and the number of functional evalutations Q.

N	M	ρ	γ	C	κ	Q
10	150	1e-3	0.90	10	9.65	3715
20	150	1e-2	0.90	10	6.39	2679
40	150	1e-2	0.90	5	3.52	2289
50	150	1e-2	0.90	3	2.35	2090
70	75	1e-4	0.90	4	1.80	2031
100	75	1e-4	0.90	2	1.27	1670

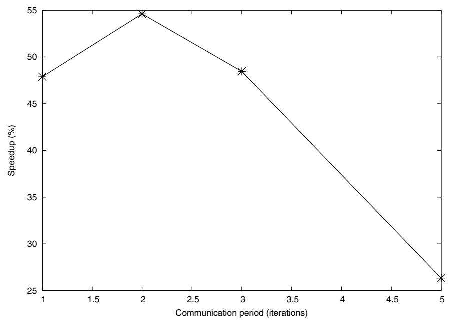


Fig. 2. Speedup vs communication period Π. The parameter values are: $N = 100$, $NP = 7$, $M = 75$, $\rho = 1e-4$, $\gamma = 0.90$.

The algorithmic parameters, such as quality criterion, number of individuals and communication period may influence the final result in a quite complicated manner. We used approximation (5) to find the number of iterations Q_s that the serial algorithm will need to reach the value of the quality functional F_p that was obtained in the parallel runs and thus to calculate the speedup and efficiency for different number of nodes. The parallel efficiency is about 80% for the 50 nodes and 55% for 100 nodes. Both speedup and efficiency vary with the number of computational nodes. For the given number of nodes, fixed population size, stopping criterion and control variable γ speedup can be plotted as function of communication period Π as shown in Figure 2 for $N = 100$, $NP = 7$, $M = 75$, $\rho = 1e - 4$, and $\gamma = 0.90$.

The reliability of the method is demonstrated by the recovery of the parameters with about 1% accuracy.

Acknowledgments

Our sincere thanks to M. Samsonova, J. Reinitz and V. Gursky for many valuable discussions. The support of the study by the NIH Grant RR07801, the CRDF GAP Awards RUB-1578-ST-05, the RFBR Grants 08-01-00315-a, 08-01-00712-a is gratefully acknowledged.

References

1. Storn, R., Price, K.: Differential Evolution. A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Technical Report TR-95-012, ICSI (1995)
2. Fan, H.-Y., Lampinen, J.: A Trigonometric Mutation Operation to Differential Evolution. Journal of Global Optimization 27, 105–129 (2003)
3. Gaemperle, R., Mueller, S.D., Koumoutsakos, P.: A Parameter Study for Differential Evolution. In: Grmela, A., Mastorakis, N.E. (eds.) Advances in Intelligent Systems, Fuzzy Systems, Evolutionary Computation, pp. 293–298. WSEAS Press (2002)
4. Zaharie, D.: Parameter Adaptation in Differential Evolution by Controlling the Population Diversity. In: Petcu, D., et al. (eds.) Proc. of 4th InternationalWorkshop on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, pp. 385–397 (2002)
5. Tasoulis, D.K., Pavlidis, N.G., Plagianakos, V.P., Vrahatis, M.N.: Parallel differential evolution. In: Congress on Evolutionary Computation (CEC 2004), Portland, Oregon (2004)
6. Chu, K.-W., Deng, Y., Reinitz, J.: Parallel simulated annealing by mixing of states. Journal of Computational Physics 148, 646–662 (1999)
7. Lawrence, P.A.: The Making of a Fly. Blackwell Sci. Publ., Oxford (1992)
8. Reinitz, J., Sharp, D.: Mechanism of Formation of Eve Stripes. Mechanisms of Development 49, 133–158 (1995)
9. Jaeger, J., Jaeger, J., Surkova, S., Blagov, M., Janssens, H., Kosman, D., Kozlov, K.N., Manu, Myasnikova, E., Vanario-Alonso, C.E., Samsonova, M., Sharp, D.H., Reinitz, J.: Dynamic control of positional information in the early drosophila embryo. Nature 430, 368–371 (2004)
10. Pisarev, A., Poustelnikova, E., Samsonova, M., Reinitz, J.: FlyEx, the quantitative atlas on segmentation gene expression at cellular resolution. Nucl. Acids Res. (2008), doi:10.1093/nar/gkn717
11. Gursky, V.V., Jaeger, J., Kozlov, K.N., Reinitz, J., Samsonov, A.M.: Pattern formation and nuclear divisions are uncoupled in drosophila segmentation: Comparison of spatially discrete and continuous models. PhysicaD 197, 286–302 (2004)

Efficiency of Parallel Monte Carlo Method to Solve Nonlinear Coagulation Equation*

Mikhail Marchenko

Institute of Computational Mathematics and Mathematical Geophysics,
Siberian Division of Russian Academy of Sciences,
pr. Lavrentieva, 6, Novosibirsk, 630090, Russia
mam@osmf.sscc.ru

Abstract. A parallel Direct Simulation Monte Carlo (DSMC) algorithm to solve a spatially inhomogeneous nonlinear equation of coagulation is presented. The algorithm is based on simulating the evolution of stochastic test particles ensembles. The algorithm can be effectively implemented on parallel computers of different architectures including GRID infrastructure based on MPLS networks. A problem of minimizing the computational cost of the algorithm is considered. To implement the algorithm on GRID infrastructure we carry out preliminary simulation of an underlying network. Such simulation enables to define minimal network bandwidth necessary for efficient parallel decomposition of DSMC algorithm.

Keywords: Kinetic equation, coagulation equation, Monte Carlo method, Direct Simulation Monte Carlo Method, multiparticle ensemble, parallel algorithm.

1 Coagulation Equation

We consider a Cauchy problem for a system of spatially inhomogeneous nonlinear equations of coagulation (the system is also referred to as **coagulation equation**) [7]:

$$\begin{aligned}
&\frac{\partial c_1}{\partial t} + \mathrm{div}(\overline{v}c_1) = -c_1 \sum_{j=1}^{\infty} K(1,j)c_j, \\
&\frac{\partial c_i}{\partial t} + \mathrm{div}(\overline{v}c_i) \\
&= \frac{1}{2}\sum_{j=1}^{i-1} K(i-j,j)c_{i-j}c_j - c_i \sum_{j=1}^{\infty} K(i,j)c_j, \quad i \geq 2, \\
&c_i(0,\overline{x}) = c_i^0(\overline{x})
\end{aligned} \tag{1}$$

* The work was supported by RFBR grants No. 09-01-00639 and No. 09-01-00035, Russian Science Support Foundation's grant.

Here $c_i = c_i(t, \overline{x}),\ i = 1, 2, \ldots$ is concentration of i-mers at time t and point $\overline{x}$, $\overline{v} = \overline{v}(\overline{x})$ is a spatially inhomogeneous velocity field, $K(i, j)$ is a coagulation kernel, $c_i^0(\overline{x})$ is a concentration of i-mers at $t = 0$. We consider the equation inside time-spatial domain $\Omega \times (0, T]$, where $\Omega \subset R^3,\ T < \infty$.

We evaluate the following functionals of the equation solution:

$$\varphi_i(t) = \int_G c_i(t, \overline{x}) d\overline{x},\ G \subseteq \Omega. \tag{2}$$

Also we consider the same space integrals of spectrum moments.

2 Monte Carlo Algorithm (Single Processor Case)

To solve coagulation equation (1) we simulate sample values of test particles ensemble

$$\xi = \xi(T) = \{p_1, p_2, \ldots, p_N\},$$

where $N = N(T)$. The pair $p_k = (l_k, \overline{x}_k)$ is called a **test particle**. Here $l_k \geq 1$ is a size of the particle (integer value), $\overline{x}_k \in R^3$ is a position of the particle. Denote by N_0 an initial number of test particles at $t = 0$.

Note that a velocity variable is not included to the phase state of the test particle (cf. with the opposite case in [2]). The reason is that the velocity filed is defined for the particle system a-priori (see (1)).

Lets us split the spatial domain Ω into sufficiently small non-overlapping subdomains $\Omega_1, \Omega_2, \ldots, \Omega_S$ (they will be referred to as **interaction subdomains**). Denote by ρ_s a volume of s-th interaction subdomain. To define Monte Carlo algorithm let us consider so called 'regularized' coagulation kernel [2,4]:

$$K^\rho(p_1, p_2) = \sum_{s=1}^{S} \rho_s^{-1} h_s(\overline{x_1}) h_s(\overline{x_2}) K(l_1, l_2), \tag{3}$$

where $h_s(\overline{x})$ is an indicator function of the domain Ω_s.

Assume that there exists a majorant for the coagulation kernel:

$$K(l_1, l_2) \leq \widehat{K} < \infty.$$

Then the majorant for the 'regularized' kernel (3) is defined as follows:

$$\widehat{K}^\rho(p_1, p_2) = \widehat{K} \rho_{min}^{-1}, \tag{4}$$

where $\rho_{min} = \min\limits_k \rho_k$.

Let us split the interval $[0, T]$ into subintervals of length $\triangle t$. According to the majorant frequency algorithm [2,4] a sample value $\xi = \xi(T)$ is simulated as follows:

1. Simulating the initial distribution of particles according to the probability density $f(i, \overline{x}) = c_i^0(\overline{x})$. Thus we have initial state of the particles ensemble $\xi(0) = \{p_1, p_2, \ldots, p_{N_0}\}$.
   Set $t = 0,\ t_c = 0$.

2. Simulating a random value τ - a time between subsequent coagulation events. The value τ is exponentially distributed with the parameter
$$\widehat{\nu} = \frac{1}{2N_0} \sum_{i \neq j} \widehat{K}^{\rho}(p_i, p_j). \quad (5)$$
Set $t_c = t_c + \tau$.
If $t_c > \triangle t$ the algorithm switches to the stage 6.
3. Choosing a pair of coagulating particles (p_i, p_j) with the probability $(0.5N\ (N-1))^{-1}$.
4. Simulating a real or fictitious coagulation event for the chosen pair. A probability of fictitious event $P_f(p_i, p_j)$ is given as follows:
$$P_f(p_i, p_j) = 1,$$
provided both the particles belong to different interaction subdomains;
$$P_f(p_i, p_j) = 1 - \frac{K(l_i, l_j)\rho_{min}}{\widehat{K}\rho_s},$$
provided both the particles belong to the same interaction subdomain Ω_s. When the real coagulation event occurs, the chosen pair of particles merges into one particle and the phase state of the ensemble changes as follows:
$$(p_i, p_j) = ((l_i, \overline{x}_i), (l_j, \overline{x}_j)) \rightarrow p_i' = (l_i + l_j, \overline{x}_i), \quad N = N - 1.$$
When the fictitious coagulation event occurs, the phase state of the ensemble doesn't change.
5. Switching to the stage 2.
6. Simulating the spatial transport of all particles according to Euler method with the step size $\triangle t$:
$$\overline{x}_i' = \overline{x}_i + \triangle t \overline{v}(\overline{x}_i).$$
Set $t_c = 0,\ t = t + \triangle t$.
7. If $t \leq T$ the algorithm switches to the stage 2.

Then the functional φ can be estimated according to the formula
$$\varphi \approx \mathrm{E}_{\xi}\zeta \approx \frac{1}{L}\sum_{i=1}^{L} \zeta_i,$$
where $\zeta_i = \zeta(\xi_i)$, the notation E_{ξ} stands for the expectation with respect to distribution of ξ.

Denote by ε_{det} a deterministic error of the estimator ζ (i.e. the error of estimating φ with $\mathrm{E}_{\xi}\zeta$). Denote by ε_{stat} a statistical error of the estimator ζ (i.e. the error of estimating $\mathrm{E}_{\xi}\zeta$ with $1/L\sum_{i=1}^{L}\zeta_i$). It is known that $\varepsilon_{stat} = \gamma\sqrt{\mathrm{D}\zeta/L},\ \gamma = \mathrm{const}$.

3 Monte Carlo Algorithm (Multiprocessor Case)

We consider a case when the number of particles N is so large that a simulation of a sample ξ must be carried out only on M processors of parallel computer:

$$\xi = \{\xi^{(1)}, \xi^{(2)}, \dots, \xi^{(M)}\}.$$

To make parallel decomposition of the single processor algorithm, the computational domain $\Omega \subset R^3$ is splitted into M non-overlapping subdomains $\widehat{\Omega}_1, \widehat{\Omega}_2, \dots, \widehat{\Omega}_M$, the particles being sorted out into subdomains. These subdomains will be referred to as **processors' subdomains**. Denote by n_m a number of particles in m-th subdomain. Each subdomain will be treated by a single processor.

Let us write the majorant frequency in the following way: $\widehat{\nu} = \sum_{m=1}^{M} \widehat{\nu}_m$, where

$$\widehat{\nu}_m = \frac{1}{2N_0} \sum_{i \neq j} \widehat{K}^{\rho}(p_i, p_j), \tag{6}$$

the summation being taken over the particles belonging to $\widehat{\Omega}_m$ [2,4]. Then a parallel simulation may be carried out as follows:

1. Each processor simulates initial condition independent of other processors (see stage 1):
$$\xi^{(m)}(0) = \{p_1^{(m)}, p_2^{(m)}, \dots, p_{n_m}^{(m)}\},\ m = 1, 2, \dots, M.$$
2. On m-th processor over the step size $\triangle t$ all coagulation events (real and fictitious) are simulated independent of other processors (see stages 2-5). In simulation instead of the parameter $\widehat{\nu}$ in (5) we use parameter $\widehat{\nu}_m$ from (6) and instead of N we use n_m. At the end of immediate interval $\triangle t$ we get
$$\xi^{(m)}(i\triangle t) = \{p_1^{(m)}, p_2^{(m)}, \dots, p_{n_m}^{(m)}\},$$
$$n_m = n_m(i\triangle t),\ m = 1, 2, \dots, M, i = 1, 2, \dots, T/\triangle t.$$
3. At the end of $\triangle t$ all processors exchange particles according to Euler method (see stage 6). Thus we get the updated ensembles $\xi^{(m)}(i\triangle t),\ m = 1, 2, \dots, M$. Then the parallel algorithm switches to the stage 2.

Requirements for parallel random number streams being very strong, it is necessary to use well tested generator. It is recommended to use the generator introduced and tested in [6].

3.1 Optimal Choice of Parallel Algorithm's Parameters

While increasing the number of processors M it is reasonable to change other parameters of the algorithm in order to get better accuracy of computations. But it is necessary to change algorithm's parameters in an optimal way otherwise the computational cost may grow.

Note that

$$\xi = \xi(\overline{p}, N_0, \triangle t, \rho, M, \{\widehat{\Omega}_m\}_{m=1}^M),$$

where ρ is a typical volume of interaction subdomains, $\{\widehat{\Omega}_m\}_{m=1}^M$ is a set of processors' subdomains and $\overline{p}$ is a set of parameters corresponding to the coagulation equation (1) and the functional (2). Note that we neglect a dependence of ξ on interaction subdomains $\{\Omega_s\}_{s=1}^S$. It is possible to do it under some restrictions on the form of interaction subdomains.

An expectation of a computer time for Monte Carlo algorithm equals to $t_1 L$, where t_1 is an expectation of a computer time to simulate one sample. It follows from an equality of a deterministic error ε_{det} and a stochastic error ε_{stat} that

$$L \sim \varepsilon_{det}^{-2} \mathrm{D}\zeta,$$

where $\mathrm{D}\zeta$ is a variance of the estimator. Therefore a **computational cost** of the algorithm is in direct proportion to the value

$$C(\zeta) = t_1 \varepsilon_{det}^{-2} \mathrm{D}\zeta.$$

Note that ε_{det} and ε_{stat} don't depend on M and $\{\widehat{\Omega}_m\}_{m=1}^M$:

$$\varepsilon_{det} = \varepsilon_{det}(\overline{p}, N_0, \triangle t, \rho),\ \varepsilon_{stat} = \varepsilon_{stat}(\overline{p}, N_0, \triangle t, \rho, L).$$

Let us call the following function a **relative efficiency** of the parallel decomposition:

$$\varPhi = \frac{C(\zeta)|_{M=1}}{C(\zeta)|_{M>1}} = \frac{t_1|_{M=1}}{t_1|_{M>1}}.$$

Here while simulating the values of $\overline{p}, N_0, \triangle t, \rho$ are the same for $M = 1$ and $M > 1$.

We assume a hypothesis that the variance of the estimator has the following dependence upon the parameters of the algorithm:

$$\mathrm{D}\zeta = D(\overline{p}, N_0, \triangle t, \rho) \sim N_0^{-1} D_1(\overline{p}),$$

the variance nearly not depending on ρ and $\triangle t$. It is possible to prove this hypothesis rigorously but this proof lies beyond the framework of this paper.

Also we assume that the deterministic error has the following order of magnitude [2,4]:

$$\varepsilon_{det} = e(\overline{p}, N_0, \triangle t, \rho) \sim E_1(\overline{p}) N_0^{-1} + E_2(\overline{p}) \triangle t + E_3(\overline{p}) \rho.$$

It follows from the last formula that

$$\triangle t \sim N_0^{-1},\ \rho \sim N_0^{-1}. \tag{7}$$

Therefore

$$\mathrm{D}\zeta \sim N_0^{-1},\ \varepsilon_{det} \sim N_0^{-1},\ L \sim \varepsilon_{det}^{-2} \mathrm{D}\zeta \sim N_0.$$

Let us investigate how the function t_1 depends on its parameters. It is evident that

$$t_1 = t_1(\overline{p}, N_0, \triangle t, \rho, M, \{\widehat{\Omega}_m\}_{m=1}^M).$$

It is clear that

$$t_1 = \mathrm{E}(t^{(i)} + t^{(c)} + t^{(e)}),$$

where $t^{(i)}$ corresponds to the simulation of the initial state of particles ensemble, $t^{(c)}$ corresponds to the independent sequential simulation of coagulation events on different processors, $t^{(e)}$ corresponds to the exchange of particles between processors. Let us derive orders of magnitude of the values $t^{(i)}, t^{(c)}, t^{(e)}$.

The parallel algorithm is synchronized at $i = 1, 2, \ldots, T/\triangle t$, namely, before data exchange stage and after having finished it. Therefore

$$t^{(c)} = \sum_{i=1}^{T/\triangle t} t_i^{(c)}, \quad t_i^{(c)} = \max_{m=1,\ldots,M} t_{i,m}^{(c)},$$
$$t^{(e)} = \sum_{i=1}^{T/\triangle t} t_i^{(e)}, \quad t_i^{(e)} = \max_{m=1,\ldots,M} t_{i,m}^{(e)},$$

where $t_{i,m}^{(c)}$ is a computer time corresponding to sequential simulation of coagulation events on m-th processor over i-th time interval $\triangle t$, $t_{i,m}^{(e)}$ is a computer time corresponding to the data exchange. It is clear that in an optimal case the following relationships for each sample must hold:

$$t_{i,1}^{(c)} \approx t_{i,2}^{(c)} \approx \ldots t_{i,M}^{(c)}, \tag{8}$$
$$t_{i,1}^{(e)} \approx t_{i,2}^{(e)} \approx \ldots t_{i,M}^{(e)} \tag{9}$$

at each step $i = 1, 2, \ldots, T/\triangle t$.

Note that it is hard to formalize a choice of $\{\widehat{\Omega}_m\}_{m=1}^M$. In what follows we give some requirements for the choice of processors' subdomains.

First of all, we assume that the number of links between processors (i.e., a virtual topology of communications) is constant while changing $\triangle t$ and M. It follows that the value of $t_{i,m}^{(e)}$ doesn't depend on the number of processors.

In what follows we describe some requirements for $\{\widehat{\Omega}_m\}_{m=1}^M$ to approximate relationships (8, 9). According to [2,4] the value of $t_{i,m}^{(c)}$ has the following order of magnitude:

$$t_{i,m}^{(c)} \sim p_m \triangle t n_m, \tag{10}$$

where the constant p_m corresponds to the performance of the computer. Therefore if $\{\widehat{\Omega}_m\}_{m=1}^M$ are chosen such that

$$n_1(0) \approx n_2(0) \approx \ldots \approx n_M(0), \tag{11}$$
$$n_1(i\triangle t) \approx n_2(i\triangle t) \approx \ldots \approx n_M(i\triangle t), \;\; i = 1, 2, \ldots, T/\triangle t$$

then the relationship (8) holds. Therefore a computational load of the parallel algorithm is quite well balanced provided $\mathrm{E}t^{(e)} \approx \mathrm{E}t^{(c)}$.

We consider a case when (11) holds automatically. It means that the coagulation equation has specific properties and the parameters of the parallel algorithm are being chosen in a specific way.

While simulating the initial condition, each processor makes the same amount of computational work. Namely, m-th processor uses the same random numbers as other processors use getting test particles in turn and choosing particles belonging to $\widehat{\Omega}_m$. Therefore

$$\mathrm{E}t^{(i)} = C_i N_0. \tag{12}$$

It follows from (11) that

$$\max_{m=1,\ldots,M} t^{(c)}_{i,m} \approx t^{(c)}_{i,m^*},\ i = 1, 2, \ldots, T/\triangle t$$

for some processor number m^*. A computational time it takes to simulate the fictitious coagulation events equals to $\frac{N_0}{M^2}\frac{C_f}{\rho}\triangle t$, where $C_f = \text{const}$. For the real coagulation events a computational time equals to $\frac{N_0}{M} C_r \triangle t$, where $C_r = \text{const}$. Therefore

$$t^{(c)}_{i,m^*} \approx \frac{N_0}{M}\big(\frac{C_f}{M\rho} + C_r\big)\triangle t, \quad \mathrm{E}t^{(c)} \sim \frac{N_0}{M}\big(\frac{C_f}{M\rho} + C_r\big). \tag{13}$$

A dependence of $t^{(e)}_{i,m}$ on the parameters $N_0, \triangle t, M, \{\widehat{\Omega}_m\}_{m=1}^{M}$ is obviously quite complicated. Also it is necessary to take into account a technology of processing the requests to send and receive data by a network software. Assume that the following relationship holds:

$$t^{(e)}_i \le C_e N_0 M^r \triangle t,\ r \ge 0,\ i = 1, 2, \ldots, T/\triangle t,$$

where $C_e = \text{const}$. Considering the upper bound as the worst case of data exchange contribution we have

$$\mathrm{E}t^{(e)} \sim C_e N_0 M^r. \tag{14}$$

Let us specify the dependence of N_0 upon M in the following way:

$$N_0 = N_0' M^d,\ 0 \le d \le 1. \tag{15}$$

If we change the variables N_0, ρ for M taking into account (7) and (15) then for the case $M > 1$ we get the following relationship:

$$\begin{aligned} t_1|_{M>1} &\sim C_i N_0 + \frac{N_0}{M}(\frac{C_f}{M\rho} + C_r) + C_e N_0 M^r \\ &\sim C_i M^d + C_f M^{2(d-1)} + C_r M^{d-1} + C_e M^{d+r}. \end{aligned}$$

Similarly, for the case $M = 1$ we get the following relationship:

$$t_1|_{M=1} \sim C_i N_0 + N_0(\frac{C_f}{\rho} + C_r) \sim (C_i + C_r) M^d + C_f M^{2d}.$$

Therefore for the case $M > 1$ the computational cost has the following order of magnitude:

$$C(\zeta) \sim M^{2d+r}. \tag{16}$$

The relative efficiency of parallel decomposition has the following order of magnitude as $M \to \infty$:

- if $C_e = 0$ then $\Phi \sim M^d \to \infty$;
- if $C_e > 0,\ d < r$ then $\Phi \sim M^{d-r} \to 0$;
- if $C_e > 0,\ d = r$ then $\Phi \sim \text{const}$;
- if $C_e > 0,\ d > r$ then $\Phi \sim M^{d-r} \to \infty$.

Let us discuss the idea of optimal use of computing system resources. Computational cost's order of magnitude being quite high as $M \to \infty$, it is reasonable to set $d < 1$. Thus we get the following additional advantage: the processors of a computing system will not be fully loaded with the calculations. Then it is possible to use free computational resources to increase the number of independent samples of ξ to decrease the value of stochastic error ε_{stat}. Alternatively, it is also possible to use sample splitting technique to decrease the value of $\mathrm{D}\zeta$.

In conclusion let us note that the foregoing results were obtained for the case when he numbers of particles $n_m,\ m = 1, 2, \dots, M$ were almost equal during simulation according to (11). But we hope that in the case when the distribution of particle ensemble among processors is quite close to (11) one a behavior of relative efficiency and computational cost will be close to the above-mentioned relationships. Surely, the most complicated cases have to be investigated later on.

3.2 Implementation of Parallel Monte Carlo Algorithm on GRID Infrastructure

In what follows we give some practical advices on the implementation of the DSMC algorithm on GRID infrastructure. We assume that computers forming GRID infrastructure have different performances. Also, underlying network is thought to be MPLS one, so we can order necessary network bandwidth. It is clear taht in the case of GRID implementation the computational cost of the data exchange is greater than the computational cost of sequential computations: $\mathrm{E}t^{(c)} \ll \mathrm{E}t^{(e)}$.

Here a main question is choosing the necessary network bandwidth. It is evident that in practice the number of processors M can not be increased infinitely. So it is not clear when the relative efficiency starts showing the asymptotic behavior (as described hereinbefore). Therefore a question arises which is a minimal network bandwidth under which

$$t_1|_{M=1} = t_1|_{M=M'}, \tag{17}$$

where M' is a number of available processors.

To determine minimal network bandwidth we make preliminary computations with $M = M'$ and necessary values of $\rho, \triangle t$ and $\{\widehat{\Omega}_m\}_{m=1}^{M}$. Actually the values of L and N_0 may be quite small. At the end we have the estimates for the values

$t^{(i)}, t^{(c)}, t^{(e)}$. Actually instead of $t^{(e)}$ it is reasonable to evaluate $b^{(e)}$ - an amount of data transferred between processors (in bytes). If necessary we can scale the values of $t^{(i)}, t^{(c)}, b^{(e)}$ to the necessary value of N_0. It easy to do it because the values of $t^{(i)}, t^{(c)}, b^{(e)}$ are in direct proportion to N_0. Having all this information we can simulate a behavior of the network using special network simulator [1]. Such simulation enables to evaluate minimal network bandwidth to satisfy the condition (17).

In details this approach is described in [5,3].

References

1. Adami, D., Giordano, S., Pagano, M.: Dynamic Network Resources Allocation in Grids through a Grid Network Resource Broker. In: INGRID 2007 (2007)
2. Ivanov, M.S., Rogazinskii, S.V.: Efiicient schemes of direct statistical simulation of rarified gas flows. Mathematical Modeling 1(7), 130–145 (1989)
3. Marchenko, M., Adami, D., Callegari, C., Giordano, S., Pagano, M.: Design and Deployment of a Network-aware Grid for e-Science Applications. In: IEEE International Conference on Communications ICC 2009 (2009)
4. Marchenko, M.A.: A study of a parallel statistical modelling algorithm for solution of the nonlinear coagulation equation. Russ. J. Numer. Anal. Math. Modelling. 23(6), 597–614 (2008)
5. Marchenko, M., Adami, D., Callegari, C., Giordano, S., Pagano, M.: A GRID network-aware for efficient parallel Monte Carlo simulation of coagulation phenomena. In: INGRID 2008 (2008)
6. Marchenko, M.A., Mikhailov, G.A.: Parallel realization of statistical simulation and random number generators. Russ. J. Numer. Anal. Math. Modelling 17(1), 113–124 (2002)
7. Voloshuk, V.M., Sedunov, Yu.S.: Processes of coagulation in disperse systems. Leningrad-Gidrometeoizdat (1975)

Parallel Algorithm for Triangular Mesh Reconstruction by Deformation in Medical Applications*

Olga Nechaeva[1] and Ivan Afanasyev[2]

[1] Supercomputer Software Department
ICMMG, Siberian Branch
Russian Academy of Science
Pr. Lavrentieva, 6, Novosibirsk, 630090, Russia
nechaeva@ssd.sscc.ru
[2] Department of Mechanics and Mathematics
Novosibirsk State University
Pirogova, 2, Novosibirsk, 630090, Russia
ivan_bunin@mail.ru

Abstract. The main goal of this paper is to develop an efficient method of triangular mesh generation for physical objects which have similar geometrical structure. The method is based on deforming a high quality mesh generated over some "ideal" object into another object of the same structure with mesh quality preservation. The approach uses the Self Organizing Maps algorithm and has been applied for constructing meshes on human femur bones using the GeomBox and GeomRandom packages. A parallel deformation algorithm is implemented using MPI. The efficiency of the parallelization is about 90%.

1 Introduction

Accurate Finite Element (FE) analysis requires high quality meshes. In some fields like medicine, geophysics, materials, etc., physical objects under analysis are described by raw data measured by laser scanners, computer tomography devices, etc. [1] The first processing stage produces a high resolution but lacking necessary quality triangular mesh. Further, this low quality triangulation has to be turned into a high quality one with less number of points. This stage is usually labor-intensive and should produce meshes with the requirements of particular FE tool [2]. The paper is focused on automation of mesh construction starting from a low quality triangulation.

The automation is applicable to a series of physical objects which have similar geometrical structure. The idea proposed in this work is based on effective reusing of high quality meshes collected so far. Once generated, a high quality mesh can be deformed in a special way to fit an object of the same structure. The deformation algorithm should provide acceptable quality of the resulting mesh.

* This work was supported by the NSU-Intel Laboratory of High Performance Computing Systems (Novosibirsk State University).

This approach is especially useful, for example, in medical applications. Particularly, we applied the proposed method for mesh generation in FE analysis of a realistic femur nail bone implant system in a typical proximal femoral fracture [3]. All human femur bones have similar geometrical structure, i.e. they differ from each other only in size and local proportions. Therefore, it is possible to deform a high quality mesh over some “ideal” bone and make it fit the femur bones of trauma patients with minimum mesh quality distortion. The method could be helpful because the same procedure of FE analysis should be applied many times for a lot of patients.

The proposed mesh deformation algorithm is based on the Kohonen’s Self Organizing Maps (SOM) [4]. SOM is a stochastic algorithm which is usually applied for projection of high dimensional data onto a low dimensional space. When running the SOM algorithm, a low quality triangulation which defines the object is used only for sampling random points over the object surface (in case of surface mesh generation) and inside of it (in case of volume mesh construction). In our present work, the case of surface mesh deformation has been considered carefully. Also it has to be noted that the proposed method is not limited to using a triangulation as the physical object description. The geometry can be given, for example, by a point cloud or voxelization.

The important issue in medical applications is the speed of diagnosis making. Therefore, the significant part of the paper is devoted to parallel implementation of the mesh deformation algorithm. Due to it’s simplicity and internal parallelism, the SOM algorithm can be efficiently parallelized for any parallel computing technology.

2 General Idea of Deformation Algorithm

Let Q be an “ideal” physical object in the Euclid space with a high quality triangular mesh $Q_N = \{q_1,...,q_N\}$, where N is the number of mesh points and $q_i \in Q$, $i = 1,...,N$ are the coordinates of these points. Let G be an object having the geometrical structure being similar to Q. In this work, G is given by some surface triangulation. There is no requirements on this triangulation, because it is used only for random point generation on G. The goal is to iteratively deform the mesh Q_N to make it fit G. The resulting mesh $G_N = \{x_1,...,x_N\}$, where $x_i \in G$, $i = 1,...,N$, should satisfy a given set $\{C_1,...,C_s\}$ of quality criteria with the given tolerance percentage values $\{\theta_1^{\max},...,\theta_s^{\max}\}$. In order to measure the accuracy of approximation of G by G_N, the distance $d(G, G_N)$ can be calculated as the maximum Euclid distance from all points in G to all triangles in G_N and vise versa. The maximum acceptable value $d(G, G_N) = d_{max}$ is to be given. When applying the iterative SOM algorithm [4] for mesh deformation, initial positions $x_i(0)$ can be obtained by a linear mapping f which matches centers of Q_N and G in such a way that $x_i(0) = f(q_i)$, $i = 1,...,N$.

At each iteration t (from T_0 to T), a random point y is generated from the surface of G uniformly; among all the mesh points $x_i(t)$ the winning point $x_m(t)$ is selected, which is closest to y; and all mesh points adjust their locations according to the rule:

$$x_i(t+1) = x_i(t) + \theta_{q_m}(t, q_i)(y - x_i(t)), \tag{1}$$

where $\theta_{q_m}(t, q_i) \in [0,1]$ controls the magnitude of mesh points displacements in G while they move towards the point y, and essentially influences the quality of resulting meshes and speed of construction process [4]. At the end of the iterative process, the SOM algorithm tends to reproduce local patterns of mesh points mutual positions in Q_N. Maximum number of iterations T is selected depending on N and tolerances $\{\theta_1^{\max}, \ldots, \theta_s^{\max}\}$. The difference of our SOM application is that Q_N is unstructured.

3 Parallel Algorithm for Mesh Deformation

In order to make the proposed deformation algorithm as fast as possible, the parallelization is considered in this section. The most time consuming operations in the SOM algorithm are (1) calculation of winning point and (2) adjustment of mesh points locations $x_i(t)$. Fortunately, all mesh points in these operations are processed in the same way independently of each other. In case of distributed memory computer system, parallel algorithm for mesh deformation is following.

Let a multicomputer consist of k processors $P_0, \ldots, P_{k-1}$. The set of mesh points G_N is distributed over the processors. It has to be noted that the points can be distributed in an arbitrary order. Let $G_N^{(j)}$ be a subset of the mesh nodes stored in the processor P_j, $j = 0, \ldots, k-1$. Let also the mesh Q_N be distributed over the processors in such a way that the processor P_j contains a subset $Q_N^{(j)}$ and if $x_i(t) \in G_N^{(j)}$, then $q_i \in Q_N^{(j)}$. It is important to ensure the same sequence of random points at each processor.

Parallel Algorithm

All processors perform the following operations at each iteration $t = T_0, \ldots, T$.

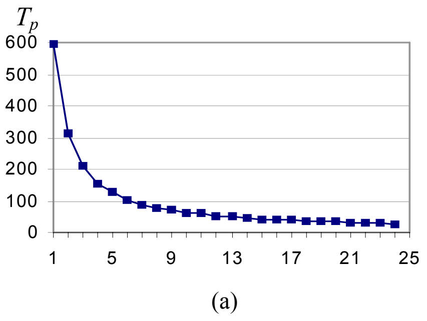
a) *Point generation.* In each processor, the same random point $y \in G$ is generated.

b) *Winner determination.* Each processor P_j searches for the point $x_{m(j)}(t) \in G_N^{(j)}$ that is the closest to y ($x_{m(j)}(t)$ is a local winner) and performs MPI_Allreduce with the operation MPI_MINLOC to distribute $\| y - x_{m(j)}(t) \|$ and determine the global winner $x_m(t)$. Then, if the local winner computed by the processor P_j is the global winner, then this processor broadcasts the point q_m to all other processors.

c) *Mesh points adjustment.* Each processor P_j adjusts locations of the mesh points by applying the rule (1) to all $x_i(t) \in G_N^{(j)}$.

It has to be pointed out that interprocessor communications occur only for global winner selection. Due to the low amount of communications, this parallel algorithm of mesh deformation is highly efficient. In case of Q_N being unstructured it is difficult to optimize the sequential algorithm of mesh deformation. It makes parallel algorithm even more valuable for unstructured meshes than structured ones.

The proposed parallel algorithm has been implemented using MPI library. In Fig.1, time has been measured for $T_0 = 0.2T$, $T = 10N$. The mesh G_N size has been equal to $N = 8034$ points. All measurements have been made in Siberian Supercomputer Center



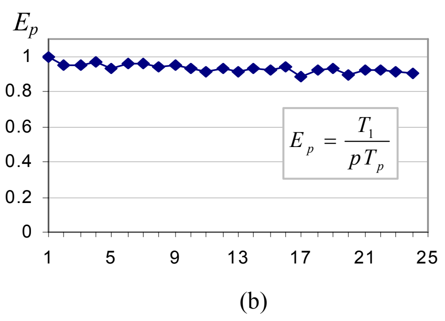


Fig. 1. Computation time (a) and efficiency (b) measured for 1 time step with T_0 = 0.2T, T = 10N and N = 8034 points

using NKS-160 system that consists of 160 processors Intel Itanium 2, 1,6 GHz. The efficiency of parallelization obtained is about 90%.

4 Application in Medicine

The proposed method of mesh deformation has been demonstrated for the problem of FE analysis of a realistic femur nail bone implant system in a typical proximal femoral fracture [3]. All human femur bones have similar geometrical structure, i.e. they differ from each other only in size and local proportions. In Fig. 2, two femur bones are shown: first is a bone of the normal man from the Visible Human Project [5], second is a bone of a trauma patient. The problem is to deform the mesh over the first bone and make it fit the second one.

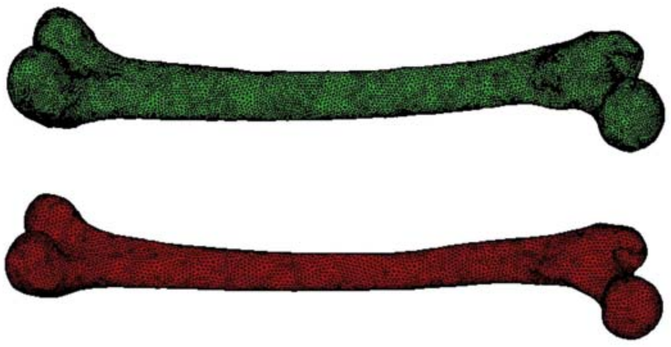


Fig. 2. Two femur bones used in computations: a bone of the normal man from the Visible Human Project (top), a bone of a trauma patient (bottom)

In order to control the resulting quality, the set $\{C_1, C_2\}$ of criteria has been used where C_1 and C_2 are the following.

1) ***C*₁ criterion.** The area of the considered triangle divided by the area of the equilateral triangle with the same circumcircle radius. The ABAQUS package [6] recommends the quality measure C_1 to be greater than 0.01.

2) ***C_2* criterion.** Minimum triangle angle: the ABAQUS package [6] recommends that the angle is to be greater than 45 degrees.

Let the tolerances for this criteria be $\theta_1^{\max} = \theta_2^{\max} = 10\%$ (depending on particular requirements). Also, let the maximum distance between G and G_N be equal to the average length of triangle edges in Q_N: $d_{max} = d_{av}(Q_N)$. After the criteria has been measured, the tolerances are calculated according to the following formula.

$$\theta_j = \frac{(C_j(Q_N) - C_j(G_N))100\%}{C_j(Q_N)}$$

According to this formula, negative values of θ_j correspond to the improvement of mesh quality, while positive values should be less than $\theta_j^{\max}$. In the table below, the results of mesh deformation are shown.

Criterion	Q_N	G_N	Tolerances
C_1	0.43322	0.397	$\theta_1 = 8.36\%$
C_2	0.39851	0.36614	$\theta_2 = 8.122\%$
d	–	–	$d(G,G_N) = 0.49d_{av}$

The quality of the deformed mesh decreased, but is still in the acceptable range. Also, the maximum distance between G and G_N satisfies the required condition.

In our implementation we used GeomRandom package by AITricks [7] which allowed us to generate random points on triangulated surface. Also, the visualization package GeomBox [7] has been used for visualizing the femur bone and all other required data.

References

1. Pursiainen, S., Hakula, H.: A High-order Finite Element Method for Electrical Impedance Tomography. PIERS Online 2(3), 260–264 (2006)
2. Date, H., Kanai, S., Kishinami, T., Nishigaki, I.: Mesh simplification and adaptive LOD for finite element mesh generation. In: Computer Aided Design and Computer Graphics, Ninth International Conference, 6 p. (2005)
3. Helwig, P., Faust, G., Hindenlang, U., Kröplin, B., Eingartner, C.: Finite element analysis of a bone-implant system with the proximal femur nail. Technology and Health Care 14(4-5), 411–419 (2006)
4. Kohonen, T.: Self-organizing Maps. Springer Series in Information Sciences, vol. 30, 501 p. Springer, Heidelberg (2001)
5. The Visible Human Project of the National Library of Medicine (National Institution of Health), `http://www.nlm.nih.gov/research/visible/visible_human.html`
6. Abaqus FEA, `http://www.simulia.com/products/abaqus_fea.html`
7. GeomRandom and GeomBox packages (AITricks), `http://aitricks.com`

Parallel Algorithms of Numeric Integration Using Lattice Cubature Formulas

Marat D. Ramazanov and Dzhangir Y. Rakhmatullin

Institute of Mathematic with Computing Centre of RAS,
112, Chernyshevsky str.,Ufa, Russia, 450008
Tel.: +7(347)272-59-36, +7(347)273-33-42, Fax: +7(347)272-59-36
RamazanovMD@yandex.ru
http://matem.anrb.ru

Abstract. The results of theory and applications of optimal lattice cubature formulas are described. The approximate integration program based on lattice formulas is considered. It has sufficiently high precision for complicated domains with smooth boundaries and dimensions up to 10 and high efficiency of paralleling.

Keywords: cubature formulas, approximate integration, calculus mathematics, functional analysis.

1 Introduction

Lattice cubature formulas are approximations of integrals $\int\limits_{\Omega} dx f(x)$ by linear combinations of f values in the lattice nodes

$$\{Hk|\ k \in \mathbb{Z}^n,\ H\text{— matrix } n \times\ n,\ \det H > 0\}, \quad Kf = \det H \sum_{Hk \in \Omega} c_k f(Hk).$$

Integrands f are from some Banach space B that formalizes their smoothness property. Norm of an error functional $l^{\Omega} : f \to \int\limits_{\Omega} dx f(x) - Kf$ in the conjugate space B^* defines a quality of the cubature formula.

Optimal formula has coefficients $\{c_k^0\}$ which minimize that norm:

$$\{c_k^0\} = \arg \min_{\{c_k\}} \|l^{\Omega}\|_{B^*}$$

This theory was founded by academic S.L. Sobolev [1]. Now, due to evolution of the theory we made algorithms near to optimal on all functional spaces generally used in calculus mathematics [2]-[4].

Programs calculated with the use of these algorithms are tested on supercomputers MVS-1000, MVS-15000VM and MVS-100 of the Joint Supercomputer Center RAS. It is clear that we can easily parallelize the lattice cubature formulas' algorithm by distributing of lattice nodes among processors of computation system. Therefore, we have high efficiency coefficient (about 70%–90%) for these programs.

Big part of communication between processors decreases performance in calculating of mathematical physics tasks. Probably it is because of “stable heredity”, that is almost all algorithms were written in single-processor computers’ age. Nowadays these algorithms are parallelized somehow. Hence we believe now one should use computational algorithms that are originally well parallelized. For example one should reduce problems to calculating of integral equations solving by using of iteration methods.

Mathematical idea of our algorithms is in reducing of stated input parameters’ smoothness (boundaries of integration domains, integrands, kernels of integral operators) into high precision of approximated numeric solutions. For example let integration domain Ω has a boundary Γ M times continuously differentiable, and integrand also belongs to $C^M(\overline{\Omega})$ class. So lattice cubature formulas with the nodes $\{hk|\ k \in \mathbb{Z}^n\}$ approximate integrals with accuracy $O(h^m),\ h \to 0$ on every space $W_p^m,\ C^m,$ with $m < M$. Moreover, for every mentioned space and for smoothness parameter M we have the same algorithm (and corresponding program) which is asymptotically optimal. It means

$$\frac{\|l_N^{\Omega,as}\|_{B^*}}{\min\limits_{c_k} \|l_N^{\Omega}\|_{B^*}} \to 1 \text{ with } N \to \infty,$$

where N is amount of nodes of cubature formula. That is the algorithm has property of conditional unsaturablity for smoothness $m < M$.

Let’s schematically describe one of our algorithms of approximate integral calculating on multidimensional bounded domains with smooth boundaries by cubature formula with nodes on the lattice $\{hk|\ k \in \mathbb{Z}^n,\ h << 1\}$. Let’s use generalized functions terminology. $l_N^{\Omega}(x) = \sum\limits_{j=0}^{J} \varphi_j(x) l_{j,N}(x)$, where $\{\varphi_j(x)\}_{j=0}^J$ is a unity partition in Ω, $\sum\limits_{j=0}^{J} \varphi_j(x) \equiv 1$, subordinated to conditions

$$\text{supp } \varphi_0 \subset \Omega,\ \forall j = \overline{1,J} \text{ supp } \varphi_j \cap \Gamma =$$

$$= \{x|\ \exists k_j \in \overline{1,n},\ x_{k_j} = \psi_j(x_1, \dots, x_{k_j-1}, x_{k_j+1}, \dots, x_n),\ \psi_j \in C^M\}\}.$$

$$l_{j,N}(x) = \chi_\Omega \cap \text{ supp } \varphi_j(x) - \det H_N \sum_{hk \in \Omega} c_{j,k} \delta(x - hk), \forall j = \overline{0,J},$$

$$\|l_{j,N}(x)\|_{(W_p^m)^*} = O(h^m), \quad \forall m < M,\ \forall p < 1.$$

To provide the last estimate we represent $l_{j,N}$ as a result of change of variables $y_s = x_s,\ s \neq k_j,\ y_{k_j} = x_{k_j} - \psi_j$ in the functional $l_j(y) = \sum\limits_{\substack{k \in \mathbb{Z}^n \\ k_j \geq 0}} \lambda_M(\frac{y-hk}{h})$.

Here $\lambda_M(y) = \chi_Q(y) - \sum\limits_{\substack{s \in \mathbb{Z}^n \\ |s| \leq S}} a_s \delta(y - s)$, $Q = [0,1)^n,\ (\lambda_M(y), y^\alpha) = 0\ \forall |\alpha| < M$.

λ_M is called elementary functional of order M. Then we recalculate integrands values in the “curved” lattice nodes $x_s^{(k)} = hk_s$ with $s \neq k_j$, $x_{k_j}^{(k)} = hk_j - \psi_j$

by values in the sought lattice $\{hk\}$. $l_j(y)$ has $(W_p^m)^*$–norm of order $O(h^m)$, $\forall m < M$. We change variables and recalculate their values from given nodes to another ones in such a way to save this order for functionals $l_{j,N}$. At last we get error functional l_N^{Ω} with $(W_p^m)^*$–norm of order $O(h^m)$ for all $p \in (1, \infty)$ and $m \in (\frac{n}{p}, M)$.

Constructed functional has bounded boundary layer (BBL-property), that is all its coefficients are bounded uniformly on h and are equal to one for the nodes that are more than Lh away from the boundary with some constant L. We proved equivalence of order and asymptotic optimality for such cubature formulas.

Theorem. *Cubature formula with BBL is asymptotically optimal on every space from the set*

$$\left\{\widetilde{W}_p^m(\Omega))\right\}_{\substack{m \in (m_1, m_2),\\ p \in (p_1, p_2)}}$$

if and only if it is optimal by order on each of them with $\frac{n}{p} < m_1 < m_2 < M,\ 1 < p_1 < p_2 < \infty.$

Besides for BBL-formulas constructed according to the algorithm we also established asymptotic optimality for the spaces W^{μ} with norms

$$\|f\|_{W^{\mu}} = \int d\xi |\tilde{f}(\xi)\mu(2\pi i\xi)|,$$

where $\tilde{f}(\xi)$ is Fourier transformation of function f, and for the Hilbert spaces $\tilde{W}_2^{\mu}$ with norms

$$\|f\|_{\tilde{W}_2^{\mu}} = \left(\sum_{k\in\mathbb{Z}^n} |f_k\mu(2\pi ik)|^2\right)^{1/2}.$$

Here we assume that function μ is infinitely differentiable, has growth estimates in infinity for $\xi \in \mathbb{R}^n$

$$C_1(1+|\xi|)^{m_1} \le |\mu(2\pi i\xi)| \le C_2(1+|\xi|)^{m_2}, \quad m_1 < m_2 < M,$$

and appears the symbol of hypoelliptic pseudodifferential operator

$$\mu(D)f(x) = \int d\xi \tilde{f}(\xi)\mu(2\pi i\xi)e^{2\pi ix\xi}.$$

Last formula is equivalent to the following estimates

$$\exists\rho > 0 \ \ \forall\alpha \ \ \exists C_\alpha \ \ \forall\xi \ \ |D^\alpha\mu(2\pi i\xi) \le C_\alpha(1+|\xi|)^{m_2-\rho|\alpha|}.$$

2 Algorithm and Program of Numeric Calculation of Integrals in n-Dimensional Case

The program "CubaInt" is designed for calculation of multidimensional integrals on bounded convex domains with smooth boundaries. Here the results of tests with some parameters [5].

1. n from 2 to 10.
2. $f(x) = \sum_{i=1}^{n} a_i x_i^{b_i}$.
3. M from 2 to 6.
4. $h = \tilde{N}^{-1}$, $\tilde{N} = 10 \ldots 10^5$.
5. $\Omega = \{x : \Phi(x) = 0,\ \Phi(x) = 1 - \sum_{i=1}^{n} c_i (x_i - 0.5)^{d_i}\}$.
6. P from 1 to 1000.

For example let's take $a = (2, 1, 2, 1, ..., 2, 1)$, $b = (2, 4, 2, 4..., 2, 4)$, $c = (6.25, 39.0625, ..., 6.25, 39.0625)$, $d = (2, 4, 2, 4..., 2, 4)$.

Calculation accuracy is estimated by decimal digits stability in answers with h tends to zero. The independent parameters are dimension n, smoothness M, amount of lattice nodes $\tilde{N}$ and the number of proccesors P.

Table 1. n=2, theoretical and experimental error orders

	experiment					theory				
$\tilde{N}\backslash M$	**2**	**3**	**4**	**5**	**6**	**2**	**3**	**4**	**5**	**6**
50	3	3	2	2	1	4	6	7	9	11
100	4	4	3	2	2	4	6	8	10	12
200	7	5	4	5	3	5	7	10	12	14
400	8	9	11	7	7	6	8	11	14	16
800	8	10	12	13	14	6	9	12	15	18
1600	9	11	13	15	16	7	10	13	17	20
3200	10	12	15	16	17	8	11	15	18	22
6400	11	14	16	18	18	8	12	16	20	23
12800	12	15	17	18	17	9	13	17	21	25

Table 2. n=3, theoretical and experimental error orders

	experiment					theory				
$\tilde{N}\backslash M$	**2**	**3**	**4**	**5**	**6**	**2**	**3**	**4**	**5**	**6**
50	3	3	2	2	1	4	6	7	9	11
100	4	4	3	3	2	4	6	8	10	12
200	6	7	5	4	4	5	7	10	12	14
400	8	8	8	8	7	6	8	11	14	16
800	9	9	10	9	9	6	9	12	15	18
1600	9	10	11	11	10	7	10	13	17	20

Tables 1–3 demonstrate the orders of absolute errors obtained both in calculations and in theoretical estimates. We can see that experimental results approximately correspond to theoretical ones except two cases. First, if we have small value of N and big M then boundary layer will not be thick enough to

Table 3. n=5, theoretical and experimental error orders

	experiment					theory				
$\tilde{N} \setminus M$	**2**	**3**	**4**	**5**	**6**	**2**	**3**	**4**	**5**	**6**
25	4	3	3	3	2	3	5	6	7	9
50	4	4	4	3	3	4	6	7	9	11
75	5	4	4	3	2	4	6	8	10	12
100	5	4	4	4	3	4	6	8	10	12
125	6	5	4	4	4	5	7	9	11	13
150	7	6	5	5	4	5	7	9	11	14
175	7	7	5	5	4	5	7	9	12	14
200	7	7	6	5	5	5	7	10	12	14

include $2M$ nodes. Second, with type "long double" we cannot increase our precision more than 18 significant digits. Here we also do not take in consideration the norm of intergant.

Now let's analyse paralleling quality of the program with the following parameters:

$$S_P = \frac{T_1}{T_P}, \qquad E_P = \frac{S_P}{P},$$

where T_P is a running time on P processors.

Figure 1 shows deviations of experimental speedups S_P (dark polylines) from the ideal ones (light straight lines).

The paralleling efficiency slightly decreases with increasing of processors quantity. That fact is caused by peculiarity of distributing of computing among

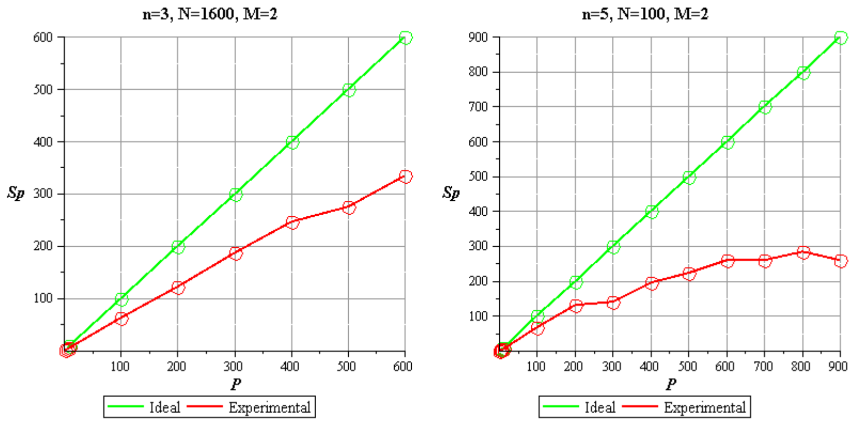


Fig. 1. Calculation speedups

processors. Every processor gets almost equal nodes quantity, but computational complexity is not equal in every node. Therefore a distribution of task is not very uniform.

References

1. Sobolev, S.L.: Introduction to the Theory of Cubature Formulas. Nauka, Moscow (1974) (in Russian)
2. Ramazanov, M.D.: To the L_p-theory of Sobolev Formulas. Siberian Advances in Mathematics 9(1), 99–125 (1999)
3. Ramazanov, M.D.: Optimization of the Lattice Cubature Formula Error Functional Norm in Scale of Wiener Spaces. Reports of Russian Academy of Sciences 36(6), 743–745 (1997) (in Russian)
4. Ramazanov, M.D.: The Cubature Formulas of S.L. Sobolev: The Evolution of the Theory and Applications. In: International Conference on Multivariate Approximation, Haus Bommerholz, p. 33. Tech. Univ. Dortmund (2008)
5. Rakhmatullin, D.Y.: Integration on Multidimensional Spaces on Multiprocessing Calculating Systems. Vychislitel'nye tehnologii 11(3), 118–125 (2006) (in Russian)

A CA-Based Self-organizing Environment: A Configurable Adaptive Illumination Facility

Stefania Bandini[1], Andrea Bonomi[1], Giuseppe Vizzari[1], and Vito Acconci[2]

[1] Complex Systems and Artificial Intelligence (CSAI) research center
Department of Computer Science, Systems and Communication (DISCo)
University of Milan - Bicocca
Viale Sarca 336/14, 20126 Milano, Italy
{bandini,bonomi,vizzari}@disco.unimib.it
[2] Acconci Studio
20 Jay St., Suite #215, Brooklyn, NY 11201, USA
studio@acconci.com

Abstract. The term Ambient Intelligence refers to electronic environments that are sensitive and responsive to the presence of people; in this paper an example of ambient intelligence whose goal is to enhance the experience of pedestrians moving inside the related physical environment will be presented. In the described approach the environment itself is endowed with a set of sensors (to perceive humans or other physical entities such as dogs, bicycles, etc.), interacting with a set of actuators that choose their actions in an attempt improve the overall experience of these users; in particular, the introduced system realizes an adaptive illumination facility. The model for the interaction and action of sensors and actuators is a dissipative multilayered cellular automata, supporting a self-organization of the system as a response to the presence and movements of people inside it. The paper will introduce the model, as well as the results of simulations of its application in a concrete case study.

1 Introduction

Ambient Intelligence [1] is a vision of future human environments endowed with a large number of electronic devices, interconnected by means of wireless communication facilities, able to perceive and react to the presence of people. These facilities can have very different goals, from explicitly providing electronic services to humans accessing the related environment through computational devices (such as personal computers or PDAs), to simply providing some form of ambient adaptation to the users' presence (or voice, or gestures), without requiring him/her to employ a computational device. Besides the specific aims of the ambient intelligent system, there is a growing interest on approaches, models and mechanisms supporting forms of self-organization and management of the components (both hardware and software) of such systems. The latter are growingly viewed in terms of autonomous entities, managing internal resources and interacting with surrounding ones so as to obtain the desired overall system behaviour as a result of local actions and interactions among system components. Examples of this kind of approach can be found both in relatively traditional pervasive computing applications (see, e.g., [2]), but also in a new wave of systems developed in the vein of amorphous computing [3] such as the one on paintable computers described in [4]. In this extreme application a whole display architecture is composed of autonomous and interacting graphic systems, each devoted to a single pixel, that must thus interact and coordinate their behaviours even to display a simple character.

This paper describes a Cellular Automata based approach to the modeling and realization of a self-organizing ambient intelligence system; the latter is viewed in terms of cells comprising sensors and actuators. The former can trigger the behaviours of the latter, both through the interaction of elements enclosed in the same cell and by means of the local interaction among adjacent cells. The transition rule adopted for the CA was derived by previous applications to reproduce natural phenomena such as percolation processes of pesticides in the soil, in specific percolation beds for the coffee industry and for the experimentation of elasticity properties of batches for tires [5,6], by modeling mechanisms of reaction and diffusion. In this specific application this rule is used to manage the interactions of cells arranged through a multilayered architecture [7], better suited to represent an artificial environment comprising a set of sensors that perceive the presence of humans (or other physical entities such as dogs, bicycles, cars), and actuators that choose their actions in an attempt improve the overall experience of these users. Throughout the paper we will adopt the example of an adaptive illumination facility, that is being designed and realized by the Acconci Studio in Indianapolis.

The developed model is the core component of an overall system supporting the design and definition of the above introduced facilities, through the simulation and envisioning of its dynamic behaviour related to specific parameters (both related to the transition rule of the CA and the number of lights and sensors). Part of the simulator generates patterns of movement of pedestrians that represent inputs for the CA and another part of the system generates a visualization of the system dynamics, interpreting the states of the CA. These parts could be actually removed and the system could be directly interfaced to field sensors and actuators, effectively piloting them, in a centralized approach. Alternative and more adequate distributed hardware/software architectures could be employed, such as in the aforementioned approaches; nevertheless the CA and its transition rule represent a formal and executable specification of the behaviour of system components.

The following section will introduce the specific scenario in which this research effort is set, describing the requirements for the adaptive illumination system and the environment adaptation model. Section 3 introduces the modeling approach, setting it in the relevant literature, while section 4 describes the developed model in details. A description of the developed environment supporting designers will follow, then conclusions and future works will end the paper.

2 The Application Scenario

The Acconci Studio was founded in 1988 to help realize public-space projects through experimental architecture and public art afforts. The method of Acconci Studio is on the one hand to make a new space by turning an old one inside-out and upside-down; and on the other hand to insert within a site a capsule that grows out of itself and spreads into a landscape. They treat architecture as an occasion for activity; they make spaces fluid, changeable, portable. They have recently completed a person-made island in Graz, a plaza in Memphis, a gallery in NY, a clothing store in Tokyo; they are currently working on a building façade in Milan, a park on a street median in Vienna, and a skate park in San Juan[1].

The Studio has recently been involved in a project for the renovation of a tunnel in the Virginia Avenue Garage in Indianapolis. The tunnel is currently mostly devoted to cars, with relatively limited space on the sidewalks and its illumination is strictly functional. The planned renovation for the tunnel comprises a set of interventions along the direction defined by the following narrative description of the project:

> The passage through the building should be a volume of color, a solid of color. Its a world of its own, a world in itself, separate from the streets outside at either end. Walking, cycling, through the building should be like walking through a solid, it should be like being fixed in color.
>
> The color might change during the day, according to the time of day: pink in the morning, for example, becomes purple at noon becomes blue, or blue-green, at night. This world-in-itself keeps its own time, shows its own time in its own way.
>
> The color is there to make a heaviness, a thickness, only so that the thickness can be broken. The thickness is pierced through with something, theres a sparkle, its you that sparkles, walking or cycling though the passage, this tunnel of color. Well no, not really, its not you: but its you that sets off the sparkle a sparkle here, sparkle there, then another sparkle in-between one sparkle affects the other, pulls the other, like a magnet a point of sparkle is stretched out into a line of sparkles is stretched out into a network of sparkles.
>
> These sparkles are above you, below you, they spread out in front of you, they light your way through the tunnel. The sparkles multiply: its you who sets them off, only you, but – when another person comes toward you in the opposite direction, when another person passes you, when a car passes by some of these sparkles, some of these fire-flies, have found a new attractor, they go off in a different direction.

The above narrative description of the desired adaptive environment comprises two main effects of illumination, also depicted in a graphical elaboration of the desired visual effect shown in Figure 1:

– an overall effect of uniformly coloring the environment through a background, ambient light that can change through time, but slowly with respect to the movements and immediate perceptions of people passing in the tunnel;
– a local effect of illumination reacting to the presence of pedestrians, bicycles, cars and other physical entities.

The first type of effect can be achieved in a relatively simple and centralized way, requiring in fact a uniform type of illumination that has a slow dynamic. The second

[1] http://www.acconci.com

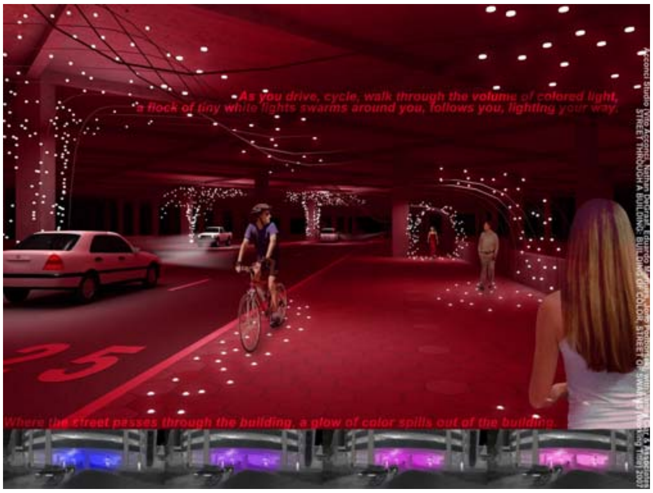

As you drive, cycle, walk through the volume of colored light, a flock of tiny white lights swarms around you, follows you, lighting your way.
Where the street passes through the building, a glow of color spills out of the building.

Fig. 1. A visual elaboration of the desired adaptive illumination facility (the image appears courtesy of the Acconci Studio)

point requires instead a different view on the illumination facility. In particular, it must be able to perceive the presence of pedestrians and other physical entities passing in it, in other words it must be endowed with sensors. Moreover, it must be able to exhibit local changes as a reaction to the outputs of the aforementioned sensors, providing thus for a non uniform component to the overall illumination. The overall environment must be thus split into parts, proper subsystems.

However, these subsystems cannot operate in isolation, since one of the requirements is to achieve patterns of illumination that are local and small, when compared to the size of the tunnel, but that can have a larger extent than the space occupied by a single physical entity ("sparkles are above you, below you, they spread out in front of you, they light your way through the tunnel"). The subsystems must thus be able to interact, to influence one another to achieve more complex illumination effects than just providing a spotlight on the occupied positions.

In the following part of the paper we will focus on this more dynamic and reactive part of the overall illumination facility. The need to consider a physical environment as an assembly of local subsystems arranged in a network, each able to decide on its own state according to a local stimulus and according to the influences of neighbouring subsystems led us to consider Cellular Automata as a suitable model to capture and reproduce the above described specification for the illumination facility.

3 Related Works

Cellular Automata (CA), introduced by John von Neumann as an environment for studying self-replicating systems [8], have been primary investigated as theoretical concept and as a method for simulation and modeling [9]. They have also been used as computational framework for specific kind of applications (e.g. image processing [10], robot path planning [11]) and they have also inspired several parallel computer architectures, such as the Connection Machine [12] and the Cellular Automata Machine [13].

Automata Networks [14] are a generalization of the classic CA, based on the introduction of the network abstraction between automata nodes. Multilayered Automata Network have been defined in [7] as a generalization of Automata Networks. The main features of the Multilayered Automata Network are the explicit introduction of a hierarchical structure based on nested graphs. Such graphs are composed of vertices and edges where each vertex can be in turn be a nested graph of lower level. A Multilayered Automata Network is directly obtained from the nested graph structure by introducing states and a transition function.

Dissipative Cellular Automata (DCA), defined in [15], are also an extension of CA. DCA differ from the basic model mainly for two characteristics: while CA are synchronous and closed systems, DCA are open and asynchronous. In particular, in DCA the cells are characterized by a thread of control of their own, autonomously managing the elaboration of the local cell state transition rule. DCA can thus be considered as an open agent system [16], in wich the cells update their state independently of each other and they are directly influenced by the environment.

In order to take advantages of both the Multilayered Automata Network and the Dissipative Cellular Automata, we introduced a new class of automata called Dissipative Multilayered Automata Network (D-MAN). An informal definition this model describes D-MAN as Multilayered Automata Network in which the cells update their state in an asynchronous way and they are open to influences by the external environment.

This extension is useful because we want to use the D-MAN as a computational environment to specify and simulate the behaviour of a distributed control system. The systems will be composed of several subsystems that are influenced by the environment in which they are situated and that are able to update their state asynchronously. Moreover each subsystem is able to communicate with its neighbours.

4 The Proposed Approach

The proposed approach adopts a Dissipative Multilayered Automata Network (D-MAN) model to realize a distributed control system able to face the challenges of the previously presented scenario. The control system is composed of a set of controllers distributed throughout the system; each of them has both the responsibility of controlling a part of the whole system as well as to collaborate with a subset of the other controllers (identified according to the architecture of the CA model) in order to achieve the desired overall system behavior. In the proposed architecture, every node is a cell of a D-MAN that can communicate only with its neighbours, it processes signals from sensors and it controls a predefined set of lights associated to it. The approach is totally distributed: there is no centralized control and no hierarchical structuring of the controllers, not only from a logical point of view but also a physical one. In the following sections, each of the components of the proposed approach will be described in details.

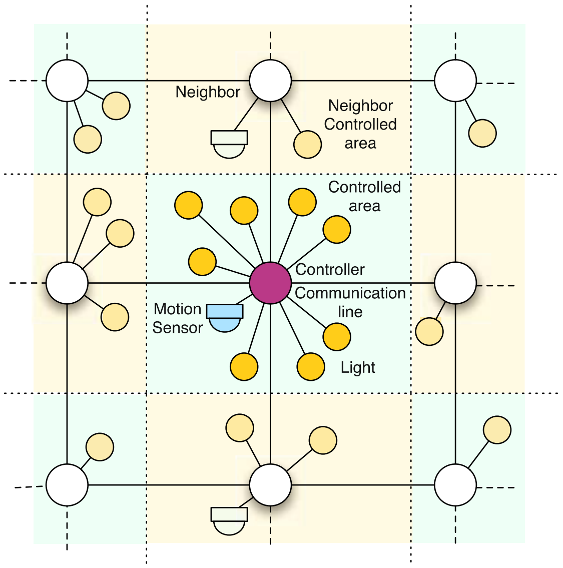


Fig. 2. The proposed architecture for the distributed control system to be managed through a D-MAN approach

4.1 System Architecture

The designed system is an homogeneous peer system, as described in Figure 2: every controller has the responsibility of managing sensors and actuators belonging to a fixed area of space. All controllers are homogeneous, both in terms of hardware and software capabilities. Every controller is connected to a motion sensor, which roughly covers the controlled area, some lights (about 40 LED lights) and neighbouring controllers.

Figure 3 describes the multiple layers of the model: the external one (level 2) is the communication layer between the controllers of the system. Every controller is an automata network of two nodes, one node is a sensor communication layer and represents a space in which every sensor connected to the microcontroller has a correspondent cell. The other node represents the actuators' layer in which the cells pilot the actuators (lights, in our case). Since the external layer is a physical one and every cell is an independent microcontroller, it cannot be assumed that the entrire network is synchronized. In same cases, a synchronous network can be constructed (for example, a single clock devices can be connected to each microcontroller or the microcontrollers can be synchronized by a process communicating with a master node), but the most general case is an asynchronous network.

4.2 Sensors Layer

The Sensor Layer is a Level 0 Dissipative Automata. As previously introduced, it is composed of a single cell, since only one sensor is connected to each microcontroller.

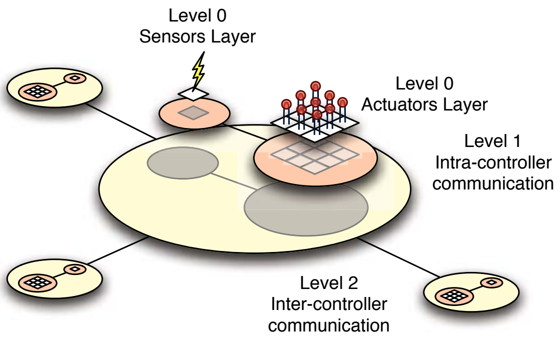


Fig. 3. The proposed automata network for the D-MAN

It is a Dissipative Automata because the internal state of the cell is influenced by the external environment. The state of the cell is represented by a single numerical value $v_s \in \mathbb{N}_{8bit}$, where

$$\mathbb{N}_{8bit} \subset \mathbb{N}_0, \forall x : x \in \mathbb{N}_{8bit} \Rightarrow x < 2^8$$

The limit value was chosen for performance reasons because 8-bit microcontrollers are widely diffused and they can be sufficiently powerful to manage this kind of situation. The value of v_s is computed as

$$v_s(t+1) = v_s(t) \cdot m + s(t+1) \cdot (1-m)$$

where $m \in \mathbb{R}, 0 \leq m \leq 1$ is the *memory coefficient* that indicates the degree of correlation between the previous value of v_s and the new value, while $s(t) \in N_{8bit}$ is the reading of the sensor at the time *s(t)*. If the sensor is capable of distance measuring, *s(t)* is inverse proportional to the measured distance (so, if the distance is 0, the value is 255, if the distance is ∞ the value is 0). If the sensor is a motion detector sensor (it able to signal 1 if an object is present or 0 otherwise) *s(t)*, s(t) is equal to 0 if there is not detected motion, *c* in case of motion, where $c \in \mathbb{N}_{8bit}$ is a constant (in our tests, 128 and 192 are good values for *c*).

4.3 Diffusion Rule

The diffusion rule is used to propagate the sensors signals throughout the system. At a given time, every level 2 cell is characterized by an intensity of the signal, $v \in \mathbb{N}_{8bit}$. Informally, the value of v at time $t+1$ depends of the value of v at time t and on the value of $v_s(t+1)$, to capture both the aspects of interaction with neighbouring cells and the memory of the previous external stimulus caused by the presence of a physical entity in the area associated to the cell.

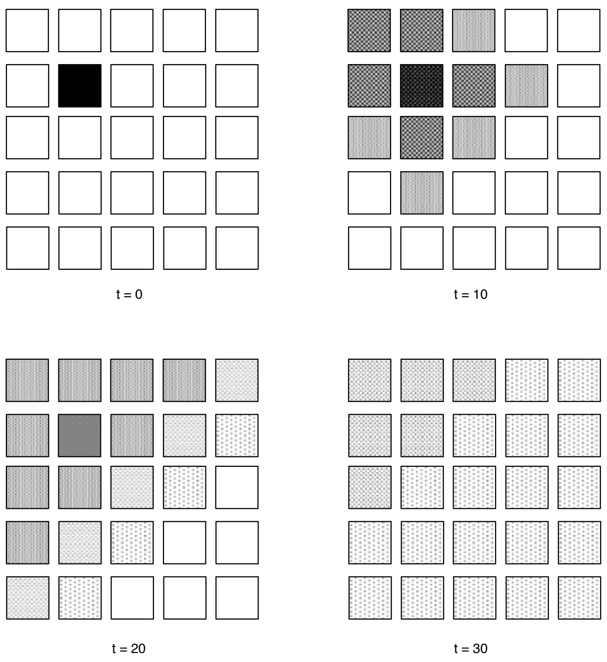


Fig. 4. An example of the dynamic behaviour of a diffusion operation. The signal intensity is spread throughout the lattice, leading to a uniform value; the total signal intensity remains stable through time, since evaporation was not considered.

The intensity of the signal decreases over time, in a process we call evaporation. In particular, let us define $\epsilon_{evp}(v)$ as the function that computes the quantity of signal to decrement from the signal and is defined as

$$\epsilon_{evp}(v) = v \cdot e_1 + e_0$$

where $e_0 \in \mathbb{R}^+$ is a constant evaporation quantity and $e_1 \in \mathbb{R}, 0 \leq e_1 \leq 1$ is the evaporation rate (e.g. a value of 0.1 means a 10% evaporation rate).

The evaporation function $evp(v)$, computing the intensity of signal v from time t to $t+1$, is thus defined as

$$evp(v) = \begin{cases} 0 & \text{if } \epsilon_{evp}(v) > v \\ v - \epsilon_{evp}(v) & \text{otherwise} \end{cases}$$

The evaporation function is used in combination with the neighbours' signal intensities to compute the new intensity of a given cell.

The automaton is contained in the finite two-dimensional square grid $\mathbb{N}^2$. We suppose that the cell $C_{i,j}$ is located on the grid at the position *i, j*, where $i \in \mathbb{N}$ and $j \in \mathbb{N}$. According to the von Neumann neighbourood [17], a cell $C_{i,j}$ (unless it is placed on the border of the lattice) has 4 neighbours, denoted by $C_{i-1,j}$, $C_{i,j+1}$, $C_{i+1,j}$, $C_{i,j-1}$.

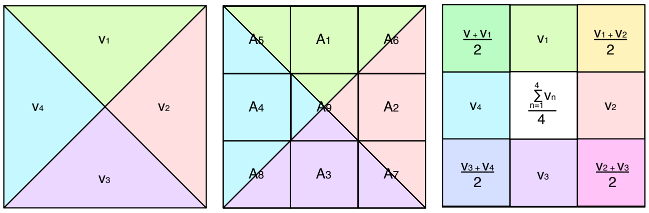


Fig. 5. Correlation between the upper layer cell subparts and the actuators layer cells

For simplicity, we numbered the neighbours of a cell from 1 to 4, so for the cell $C_{i,j}$, N_1 is $C_{i-1,j}$, N_2 is $C_{i,j+1}$, N_3 is $C_{i+1,j}$, and N_4 is $C_{i,j-1}$

At a given time, every cell is characterized by an intensity of the sensor signal. Each cell is divided into four parts (as shown in Figure 5), each part can have a different signal intensity, and the overall intensity of the signal of the cell is the sum of the parts intensity values. The state of each cell $C_{i,j}$ of the automaton is defined by $C_{i,j} = \langle v_1, v_2, v_3, v_4 \rangle$ where $v_1, v_2, v_3, v_4 \in \mathbb{N}_{8bit}$ represent the intensity of the signal of the 4 subparts. $V_{i,j}(t)$ represents the total intensity of the signals (i.e. the sum of the subparts signal intensity) of the cell i, j at time t. The total intensity of the neighbours are denoted by V_{N1}, V_{N2}, V_{N3}, and V_{N4}. The signal intensity of the subparts and the total intensity are computed with the following formulas:

$$v_j(t+1) = \begin{cases} \frac{evp(V(t))\cdot q + evp(V_{Nj}(t))\cdot(1-q)}{4} & \text{if } \exists N_j \\ \frac{evp(V(t))}{4} & \text{otherwise} \end{cases}$$

$$V(t+1) = \sum_{i=1}^{4} v_i(t+1)$$

where $q \in \mathbb{R}, 0 \leq q \leq 1$ is the conservation coefficient (i.e. if q is equals to 0, the new state of a cell is not influenced by the neighbours values, if it is equals to 0.5 the new values is a mean among the previous value of the cell and the neighbours value, if it is equals to 1, the new value does not depend on the previous value of the cell but only from the neighbours). The effect of this modeling choice is that the parts of cells along the border of the lattice are only influenced through time by the contributions of other parts (that are adjacent to inner cells of the lattice) to the overall cell intensity.

4.4 Actuators Layer

The cells of the actuator layer determinate the actuators actions. In this project the actuators are LED lamps that are turned on and off according the the state of the cell. Instead of controlling a single LED from a cell, every cell is related to a group of LEDs disposed in the same (small) area.

In the case of regular neighbourhood, each controlled area in divided into 9 sub-areas and each sub-area contains a group of LEDs controlled by the same actuators layer cell. The state of each cell is influenced only by the state of the signal intensity of the upper layer cell. The correlation between the upper layer cell subparts and the actuators layer cells is shown in Figure 5.

The state of the actuators cells $A_1..A_9, A_j \in N_{8bit}$ is computed with the following formula:

$$A_i(t+1) = \begin{cases} v_i(t+1) & 1 \le i \le 4 \\ \dfrac{v_4(t+1)+v_1(t+1)}{2} & i=5 \\ \dfrac{v_1(t+1)+v_2(t+1)}{2} & i=6 \\ \dfrac{v_2(t+1)+v_3(t+1)}{2} & i=7 \\ \dfrac{v_3(t+1)+v_4(t+1)}{2} & i=8 \\ \dfrac{1}{4}\displaystyle\sum_{j=1}^{4} v_j(t+1) & i=9 \end{cases}$$

There are different approaches to associate LED activity (i.e. being on or off, with which intensity) to the state of the related actuator cell. A first one consists in directly connecting the lights' intensity to the signal level of the correspondent cell; more details on this will be given in the following Section.

5 The Design Environment

The design of a physical environment (e.g. building, store, square, road) is a composite activity, comprising several tasks that gradually define the initial idea into a detailed project, through the production of intermediate and increasingly detailed models. CAD softwares (e.g AutoCAD), and also 3D modelling applications (e.g. Autodesk 3DStudio Max, Blender) are generally used to define the digital models for the project and to generate photo realistic renderings and animations. These applications are extremely useful to design a lights installation like the one related to this scenario, but mainly from the physical point of view. From the obtained 3D models it is easy to extract the necessary information for a correct positioning of lights in the real space.

In order to generate a dynamics in this kind of structure, to grant the lights the ability to change illumination intensity and possibly color, it is also possible to "script" these applications in order to characterize lights with a proper behaviour. Such scripts, created as text files or with graphical logic editors[2], define the evolution of the overall system over time. These scripts are however heavily dependent on the adopted software and they are not suitable for controlling real installations, even though they can be used to achieve a graphical proof of concept. Another issue is that these tools are characterized by a "global" approach, whereas the system is actually made up of individual microcontrollers' programs acting and interacting to achieve the global desired effect.

The issue of defining a local behaviour for autonomous simple components leading to a given overall behaviour is actually a significant research problem (see, e.g., [18] for an investigation in this direction). In this experience, our aim was to facilitate the

[2] For example, Blender has a graphical logic editor for defining interactive behaviour either using a Graphical User Interface or exploiting a Python API for a more sophisticated control.

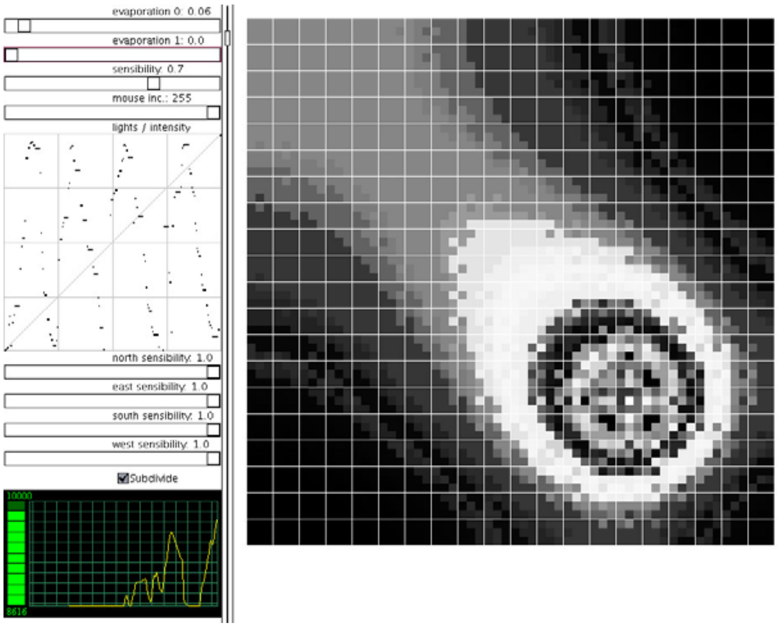

evaporation 0: 0.06
evaporation 1: 0.0
sensibility: 0.7
mouse inc.: 255
north sensibility: 1.0
east sensibility: 1.0
south sensibility: 1.0
west sensibility: 1.0
☑ Subdivide

Fig. 6. A screenshot of the design environment. On the left, there is the system configurator and the global intensity graph, on the right the lights view.

user in designing the dynamic behavior of a lights installation by supporting the envisioning of the effects of a given configuration for the transition rule guiding lights; therefore we created an ad-hoc tool, also shown in Figure 6, comprising both a simulation environment and a graphical parameters configurator. This tool support the specification of the values for some of the parameters of the transition rule, affecting the global behavior of the overall system. The integrated simulation helps understanding how the changes of the single parameters influence the overall behavior of the illumination facility: every changed parameter is immediately used in the transition rule of every cell.

In the following paragraphs, the tool's main components are described. Ad the end of this section, some experimental configurations and the related dynamic evolution are presented.

5.1 The Cells Simulator

The main component of the design environment is the simulator. This component simulates the dynamic evolution of the cell over the time, according to the transition rule. The simulated cells are disposed over a regular grid and each cell is connected to its neighbors according to the von Neumann neighbourood. By default, the tools is configured to simulate 400 cells, organized in a 20x20 grid. The grid is not toroidal, to better simulate a (portion of) the real installation space. Each cell has an internal state represented as an 8 bits unsigned number. In order to better simulate the real asynchronous system, an independent thread of control, that re-evaluates the internal state of the cell every 200 ms is associated to each cell. At the simulation startup, each thread starts after a small (< 1 s) random delay, in order to avoid a sequential activation of the threads, that is not realized in the real system. The operating system scheduler introduces additional random delays during both the activation and the execution cycle of the threads.

5.2 The Lights View

The aim of the Lights View is to realize an interactive visualization of the dynamic evolution of the system. In particular, the user can simulate the presence of people in the simulated environment by clicking on the cells and moving the mouse cursor. Each cell of the simulated system is associated an area of the screen representing a group of lights controlled by the cells. More precisely, it is possible to define at runtime if the area controlled by each cell is subdivided in 9 sub-areas (9 different lights groups) or if it is a single homogeneous light group. Each simulated group of lights is characterized by 256 different light intensity levels.

On the left of lights view, there is a graph showing the evolution over the time of the sum of all the cells intensity levels. This graph is particularly useful to set the coefficients of the evaporation function.

5.3 The System Configurator

Through this component, the user can define most of the parameters related to the transition rule of the simulated system.

The first two sliders control the evaporation coefficients e_0 and e_1, the next one controls the sensibility parameters q (see Section 4.3 for the parameters' semantics). The "mouse increment" slider defines the amount of the increment in the cell intensity when a user clicks on the cell: it represents the sensitiveness of the cell to sensor stimulus in the real system.

Under the four sliders there is a small panel that supports drawing the function that correlates the internal cell intensity value and the correspondent light group intensity value. The default function, represented by a diagonal segment between the (0,0) position and the (255,255) position, is the "equal" function (i.e. if the cell intensity has value x, the lights intensity has value x). It is possible to draw an arbitrary function, setting for each cell intensity value a correspondent light intensity value simply drawing the function over the graph.

The last four sliders control the sensitivity of each cell to the neighbors in the four directions (q_N, q_E, q_S, q_W); by keeping these values separated it is possible to configure the cell to be more sensitive to the cells in a specific direction (e.g. left or right).

Finally, there is a check-box to switch between 1 and 9 lights groups per cell.

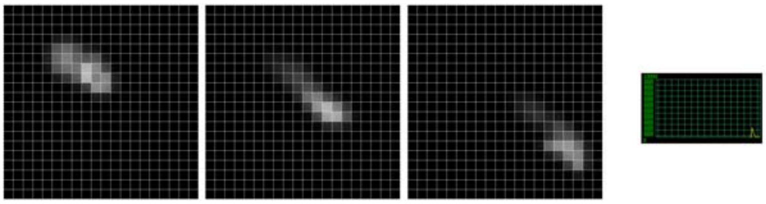


Fig. 7. Example 1: $e_0 = 0.75, e_1 = 0, q = 0.1, f = eq, q_N = q_E = q_S = q_W = 1$

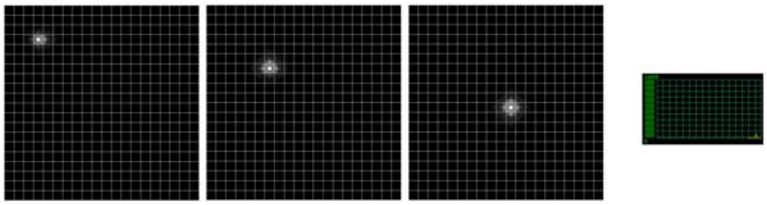


Fig. 8. Example 2: $e_0 = 1, e_1 = 0.1, q = 0.5, f = eq, q_N = q_E = q_S = q_W = 1$

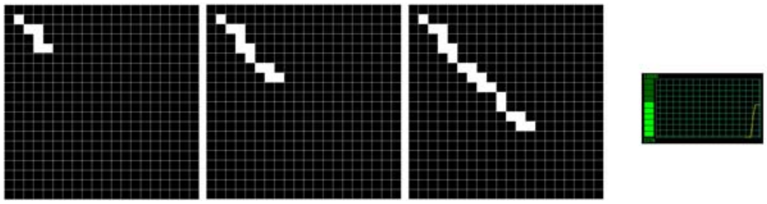


Fig. 9. Example 3: $e_0 = 0.05, e_1 = 0, q = 0.0, f = eq, q_N = q_E = q_S = q_W = 1$

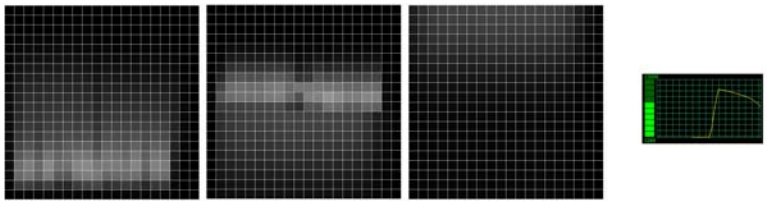


Fig. 10. Example 4: $e_0 = 0.01, e_1 = 0, q = 0.0, f = eq, q_N = 0.1, q_E = q_S = q_W = 0$

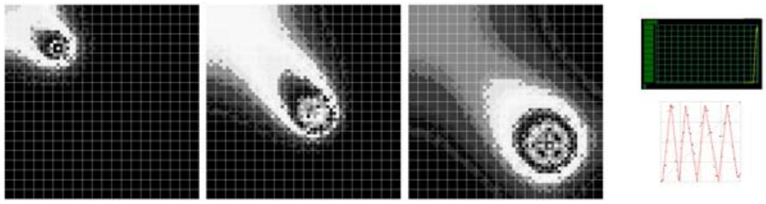


Fig. 11. Example 5: $e_0 = 0.6, e_1 = 0, q = 0.7, q_N = q_E = q_S = q_W = 1$

5.4 Experimental Configurations

This last section presents some consideration about the relations between the parameters value and the system behavior. This is not intended to be exhaustive analysis, it only presents some relevant usage examples of the design environment.

The first example, shown in Figure 7, describes 3 steps of evolution of the system configured with the default parameters ($e_0 = 0.75, e_1 = 0, q = 0.1$, mouse increment = 255, $f = eq, q_N = q_E = q_S = q_W = 1$). The system, in this configuration, acts as a sort of spot-light around the stimulated areas. When the movement of a person on the space is simulated with a mouse input a light trace is generated following the mouse movement. The trace is not present anymore in Figure 8, with an increased evaporation coefficient ($e_0 = 1, e_1 = 0.1, q = 0.5$); on the contrary, in Figure 9 a long-persistent tail is produced with a very low evaporation level and no neighbours sensibility ($e_0 = 0.05, e_1 = 0, q = 0$). Figure 10 shows a different configuration with a high sensitivity to the southern neighbour ($e_0 = 0.01, e_1 = 0, q = 0.0, f = eq, q_N = 0.1, q_E = q_S = q_W = 0$). A sort of "smoke" arising from southern cells can be viewed. The last example, shown in Figure 11, is achieved through an ad-hoc intensity-light correlation function (shown in red line in the figure) and the following parameters: $e_0 = 0.6, e_1 = 0, q = 0.7, q_N = q_E = q_S = q_W = 1$.

It is interesting to notice how many different behaviors can be achieved by means of a different parameter specification of the same transition rule.

6 Future Development

The paper introduced an ambient intelligence scenario aimed at improving the everyday experience of pedestrians and people passing through the related environment. A specific scenario related to the definition and development of an adaptive illumination facility was introduced, and a CA-based model supporting the specified behaviour for the illumination facility was defined. A prototype of a system supporting designers in the definition of the relevant parameters for this model and for the overall illumination facility was also introduced.

The renovation project is currently under development on the architectural and engineering side, whereas the CA-based model has shown its adequacy to the problem specification, both in order to provide a formal specification of the behaviour for the system components. The realized prototype explored the possibility of realizing an ad hoc tool that can integrate the traditional CAD systems for supporting designers in simulating and envisioning the dynamic behaviour of complex, self-organizing installations. It has been used to understand the adequacy of the modeling approach in reproducing the desired self-organized adaptive behaviour of the environment to the presence of pedestrians. We are currently improving the prototype, on one hand, to provide a better support for the Indianapolis project and, on the other, to realize a more general framework for supporting designers of dynamic self-organizing environments.

The modeling approach, finally, can also be adopted as a mechanism specifying and simulating the interaction of physically distributed autonomous components, for instance in monitoring and control applications.

References

1. Shadbolt, N.: Ambient Intelligence. IEEE Intelligent Systems 18(4), 2–3 (2003)
2. Filho, A.E.S., Lupu, E.C., Dulay, N., Keoh, S.L., Twidle, K.P., Sloman, M., Heeps, S., Strowes, S., Sventek, J.: Towards supporting interactions between self-managed cells. In: [19], pp. 224–236
3. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight Jr., T., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R.: Amorphous computing. Commun. ACM 43, 74–82 (2000)
4. Butera, W.: Text display and graphics control on a paintable computer. In: [19], pp. 45–54
5. Bandini, S., Erbacci, G., Mauri, G.: Implementing cellular automata based models on parallel architectures: The capp project. In: Malyshkin, V.E. (ed.) PaCT 1999. LNCS, vol. 1662, pp. 167–179. Springer, Heidelberg (1999)
6. Bandini, S., Mauri, G., Pavesi, G., Simone, C.: Parallel simulation of reaction-diffusion phenomena in percolation processes: A model based on cellular automata. Future Generation Comp. Syst. 17(6), 679–688 (2001)
7. Bandini, S., Mauri, G.: Multilayered cellular automata. Theor. Comput. Sci. 217(1), 99–113 (1999)
8. von Neumann, J.: Theory of Self-Reproducting Automata. University of Illinois Press (1966)
9. Weimar, J.R.: Simulation with Cellular Automata. Logos Verlag, Berlin (1997)
10. Rosin, P.L.: Training cellular automata for image processing. IEEE Transactions on Image Processing 15(7), 2076–2087 (2006)
11. Behring, C., Bracho, M., Castro, M., Moreno, J.A.: An algorithm for robot path planning with cellular automata. In: Bandini, S., Worsch, T. (eds.) ACRI 2000, pp. 11–19. Springer, Heidelberg (2000)
12. Hillis, W.D.: The Connection Machine. MIT Press, Cambridge (1985)
13. Margolus, N., Toffoli, T.: Cellular Automata Machines. A new environment for modelling. MIT Press, Cambridge (1987)
14. Goles, E., Martinez, S.: Neural and Automata Networks: Dynamical Behavior and Applications. Kluwer Academic Publishers, Dordrecht (1990)
15. Zambonelli, F., Mamei, M., Roli, A.: What can cellular automata tell us about the behavior of large multi-agent systems? In: Garcia, A.F., de Lucena, C.J.P., Zambonelli, F., Omicini, A., Castro, J. (eds.) Software Engineering for Large-Scale Multi-Agent Systems. LNCS, vol. 2603, pp. 216–231. Springer, Heidelberg (2003)
16. Jennings, N.R.: On agent-based software engineering. Artif. Intell. 117(2), 277–296 (2000)
17. Gutowitz, H.: Cellular Automata: Theory and Experiment. MIT Press/Bradford Books, Cambridge (1991)
18. Yamins, D., Nagpal, R.: Automated global-to-local programming in 1-d spatial multi-agent systems. In: AAMAS 2008: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems, Richland, SC, International Foundation for Autonomous Agents and Multiagent Systems, pp. 615–622 (2008)
19. Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, July 9-11. SASO. IEEE Computer Society, Los Alamitos (2007)

A Lattice-Gas Model of Fluid Flow through Tortuous Channels of Hydrophilous and Hydrophobic Porous Materials*

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 cellular automata (CA) approach is proposed for simulating a fluid flow through the porous material with tortuous channels and different wetting properties of pore walls. The approach aims to combine CA methods both for construction the structure of porous material model and to simulate the fluid flow through it. It is shown that any kind of tortuous structure may be obtained by pattern formation CA evolution, which is then used as a medium for Lattice Gas CA model application. The model is provided by special boundary conditions to account for additional tension forces between solid and liquid substances, determining the hydrophobic and hydrophilous properties of the material. The model has been tested on a small 2D array to obtain some dependencies of flow velocity on the tortuosity and wetting properties of pore walls. Parallel implementation of flow simulation through a carbon electrode of a hydrogen fuel cell is also performed, demonstrating high efficiency (>70%) of parallelization.

Keywords: cellular automata, Lattice-Gas models, pattern formation, porous medium, parallel implementation.

1 Introduction

Fluid flow in porous materials is of great interest for many reasons. Porous materials are extensively used both in engineering (building materials, petroleum recovery) and research (catalysis, electrochemistry). The main characteristic of a certain porous material is its permeability for liquids or gases. A bright example of porous materials application is a hydrogen Proton Exchange Membrane Fuel Cell, the main parts of which – the polymer membrane and the carbon electrodes are made of porous substances, power generating capability of the cell strongly depending of the materials properties [1]. Obviously, simulation of processes in

* Supported by 1) Presidium of Russian Academy of Sciences, Basic Research Program N 2 (2009), 2) Siberian Branch of Russian Academy of Sciences, Interdisciplinary Project 32 (2009).

porous components is useful, and, sometimes, urgently needed both when the material is produced, and when the device is designed.

The problem of porous material simulation is not a new one. The earlier investigations are dated to the middle of the past century. At that time a porous medium was regarded as a bulk substance, whose permeability properties being characterized by a porosity coefficient (ratio of pore volume to that of solid substance). The famous Darcy low was mainly used when dealing with porous materials. At the age of computer and numerical methods a number of mathematical models based on Partial Differential Equations were developed [2], being capable to simulate the fluid flow *at pore level*, i.e.simulating the flow through the pore channels. These models experience the following difficulties. First, the construction of porous medium itself yielding in boundary conditions is a separate nontrivial problem. Usually, the porous material is represented by many randomly allocated simple geometrical figures like circles and rectangles of appropriate size. Such an approximation is sometimes too coarse, especially when soft porous materials such as polymers or carbons are under consideration. The second problem lays in the complexity of the differential equations solution. To simulate the flow the nonlinear Navier-Stokes equation is to be solved, which is a hard task due to the complex boundary conditions and difficulty of parallel implementation. To make the problem easier convection-diffusion models [2] based on linear partial differential equations is frequently used. In both cases the irregularity of pores configuration causes troubles in writing the equations. To overcome last difficulty the method referred to in [3] as "computing from images" has been proposed. This idea was implemented in the application of fine grained simulation models, such that finite element and finite difference numerical methods and cellular automata. The fine-grained explicit form of computational process representation on a lattice allows to map it onto the pixels of the material image. Moreover, in some practical cases the image may be given as digitized photo or computer tomography [3], but when a general method of simulation is under development it is worth to have special algorithms and programs for obtaining the initial image of the medium with proper pore channels configuration.

With the appearing of Lattice-Gas [4] cellular Automata hydrodynamics the investigations of porous materials at porous level (as a flow in pores) make progress rapidly, the majority of methods used being based on the Lattice-Boltzmann (LBM) method [6]. A short but complete review of LBM methods implementation for porous material study is given in [7]. Some of them are very sophisticated allowing for permeability prediction and calculating the tortuosity of the flow. Very impressive results are presented in [8,9], where 3D flows are simulated at pore level.

For all that, there remain many aspects to be studied. Some of them motivate the approach used in the paper. They are as follow.

1) Although Lattice-Boltzmann hydrodynamics is considered more advanced and is nowadays more frequently used, the classic form of Lattice Gas cellular automata has its advantages. The first is the absence of round off errors, and

the second is computational stability, both features being the consequence of the completely discrete representation of all values used in simulation process. A not the least of the factor is the desire to expand the classical approach to CA hydrodynamics, combining it with CA methods for the porous medium model construction.

2) Indeed, in the majority of studies the porous material is represented as a structure built out of rectangles, cylinders or spheres, each fitting better with a group of porous materials, the universal case being unlikely achievable. In our investigation we show preferences to Cellular Automata techniques which give us an opportunity to obtain porous medium model by exploiting the CA capability of pattern formation. The proper choice of CA parameters allows to obtain the pore model with wanted pattern motifs and tortuosity. We expect that such a model fits with soft porous materials [2] to the best advantage. So, the first objective of the paper is to introduce and test the method of obtaining the medium model using CA techniques.

3) It is quite clear, that for simulating even a small specimen a very large cellular automaton is needed. Hence, parallel implementation is unavoidable, and its efficiency should also be taken into consideration when evaluating the method features. The fact that simulation is accomplished over a composition of two CAs with identical size, makes reasonable the parallelization of both cellular CAs as a single task.

Based on these three arguments an approach to simulation of a fluid flow through porous medium is presented. The approach is constrained to static porous media, whose morphology does not change during the process under investigation. After the Introduction, in Section 2, the problem statement is given intuitively and formally. In section 3 the porous medium model and its computer representation are described. Section 4 is devoted to the boundary conditions of hydrophilous and hydrophobic porous materials. In section 5 the results of testing the method on a single computer and its implementation on a supercomputer cluster are presented.

2 The Problem Statement

Simulation of fluid flow through a porous material aims to investigate the porous material properties. Simulation methods and computer tools are in demand when the material production is under development, and also when a certain device is designed in which the flow through the porous membrane is used, or on the contrary such a flow should be prevented. Porous medium itself is a very complex entity for computer simulation. It cannot be found two porous material specimen which are quite identical, hence there is indeterminacy in its mathematical description. When pores are filled with flowing liquid the process representation becomes yet more uncertain. Hence, simulation methods can not claim to have a high degree of accuracy. Most likely, the simulation results should be regarded as qualitative ones, although some quantitative characteristics can also be obtained.

The simulation method here is developed for the 2D case. Usually 2D version is regarded as an approximation of the 3D one, the last being of actual interest. In our approach an attempt is made to develop fluid flow simulation process together with the procedure of obtaining the porous medium model. Thus, the simulation task turns out to be a superposition [10] of two cellular automata, i.e. ($\aleph_P = \langle A_P, M, \Theta_P \rangle$ and $\aleph = \aleph_F(\aleph_P)$, where $\aleph_F = \langle A_F, M, \Theta_F \rangle$ stand for porous medium model and for flow model, respectively. Accordingly, they are further referred to as a "PoreCA" and "FlowCA". The first may be either synchronous or asynchronous, the second is synchronous. They have identical naming sets $M_P = M_F = M$, but different alphabets $A_P \neq A_F$ and different transition function sets $\Theta_P \neq \Theta_F$. The composed model allows us to perform a number of simulation with slightly changed initial conditions for $\aleph_P$. In other words, to perform simulation of the same flow through several specimens slightly differing in patterns but having the same motif inherent to the type of the porous material under investigation. Performing a number of simulations on those patterns with the same motif one may obtain far more reliable information about the porous properties.

The simulation task is formulated as follows. Given are two sets of parameters. The first is concerned with the porous material properties. It is characterized by the following data.

1) The *size* of the specimen. Since this study is confined to a 2D case, the length and the width of the specimen should be known.

2) *Porosity* coefficient

$$Por = S_0/S_1, \tag{1}$$

where S_0 and S_1 are the areas occupied by pores and solids, respectively.

3) *Resistance* of medium to the flow. The notion is newly introduced here. In part, it is a reciprocal of the habitual "permeability" taking into account the impact of pore channels layout relative to the flow direction.

$$Res = L_1/L_0, \tag{2}$$

where L_1 and L_0 are the lengths of the projections of the total pore-solid border length L onto the main flow direction (L_0) and onto that one being orthogonal to it (L_1).

5) *Tortuosity* of pore channels. Although the notion is qualitative we introduce a quantitative assessment, which is expressed through the amounts of extremities in the "pore border lines" $b(x, y)$, i.e.

$$Tort = E_{extr} \tag{3}$$

where E_{extr} is a number of extremities of the pore border lines allocated on a unit of area.

6) *Wettability.* This property characterizes the interfacial tension between the liquid and solid. It has a quantitative measure as a *wetting angle*, which depends on the solid and liquid substances properties and smoothness of pore surface. The accuracy of that measure is poor, moreover, for most porous materials it is

not known. Hence, we confine ourselves to qualitative estimate in terms of the words *hydrophobic* (not wettable), and *hydrophilous*(wettable).

Moreover, the porous material 2D representations differ by pattern images. In order to differentiate pore media by image types we follow [11], where the patterns are classified according to the associations with common life pictures, which are referred to as "motifs", such as "patches", "stripes" "clouds", "islands".

The second set of given data includes the liquid substance properties: density and viscosity. The fluid is assumed to inflow from one side (say, the left one) of the specimen and to outflow on the opposite side, the pressure drop or the mean velocity of the flow being given. These data are used for constructing the FlowCA and for giving physical interpretation to the simulation results [15].

Each simulation run aims to produce the following information.

1) The flow rate Q, i.e. the amount of liquid passing through the specimen in a unit of time .

2) The flow velocity distribution $V(i,j)$ in pores (velocity field, maximum velocity, existence of vortices, cavities).

A series of simulation runs allows to obtain some useful dependencies of Q and $V(i,j)$ on pressure drop, channel tortuosity, wettability of solid material.

3 CA Models of Porous Media

A class of cellular automata which simulates pattern formation may be successfully used to obtain the model of porous medium. It is a class 2 according to Wolfram's classification [12]. It comprises CAs which in their evolution tend to a stable global configuration. With Boolean alphabet of the CA the cell states equal to "ones" are usually represented in black, and those equal to "zero" – in white, forming a kind of tortuous patterns, looking like curved stripes, fanciful patches, spirals, ovals, which are classified as "motifs". The motifs and the degree of tortuosity may be regulated by changing the CA parameters and initial global state. Hence, using the pattern formation CAs it is possible to obtain the wanted porous medium model in the form of an image where solids are in black and the empties are in white.

Two types of pattern formation CA are the most suitable for being used for porous medium model construction. They are as follows.

1) The so called "phase separation" CA [13], and

2) The CAs with weighted templates, which are more known as Cellular Neural Networks (CNN) [11].

To represent the PorCA $\aleph_P = \langle A, M, \theta \rangle$ formally it is enough to specify its *alphabet*, which is a Boolean one $A_P = \{0,1\}$, its size $I \times J$, which determine the set of cell coordinates $M_P = \{(i,j) : i = 0,\ldots,I; j = 0,\ldots,J\}$ referred to a set of *cell names*. A pair $(a,(i,j)) : a \in A, (i,j) \in M$, is referred to as a *cell,* and the set of CA cells is called as a *cellular array* $\Omega = \{(a,(i,j)) : A \in A; i = 0,\ldots,I; j = 0,\ldots,J)\}$.

The third notion is a *local operator* denoted by θ. Local operator is spatially translational. It is given in a form of substitution, which changes some states of a set of closely allocated cells, called further according to [14] as a *local configuration*

$$S(i,j) = \{(v_0,(i,j)),...,(v_k,\phi_k(i,j)),...,(v_q,\phi_q(i,j))\}, \tag{4}$$

where $V_S = \{v_0, v_1, ..., v_q\}$ is a *state template* of $S(i,j)$, and

$$T_S = \{(i,j), \phi_1(i,j), ..., \phi_k(i,j), ...\phi_q(i,j)\} \tag{5}$$

is the *naming template* for $S(i,j)$. The functions $\phi_k(i,j)$ in (5) indicate the cell names that form a *neighborhood* for the cell (i,j).

A local operator consists of two local configurations with a substitution symbol in between.

$$\theta(i,j) : S(i,j) \rightarrow S'(i,j), \tag{6}$$

where $S(i,j)$ and $S'(i,j)$ are referred to *as a basic,* and a *next state* local configurations of θ, respectively, (i,j) being called a *main cell* for θ, the corresponding state templates being $V_S = \{v_0, v_1, ..., v_q\}$ and $U_S = \{v'_0, ..., v'_r\}$, respectively, $r \leq q$.

Next states $v'_k \in V'_{S'}$, $k = 0, ..., r$, are values of a *transition function*

$$v'_k = f_k(v_0, ..., v_q), \qquad k = 0, 1, ..., r. \tag{7}$$

In PorCAs transition functions are either Boolean functions or simple arithmetical ones.

Application of θ to a cell $(i,j) \in M$ consists of two actions:

1) computing next states according to (7), and

2) updating cells of $S(i,j)$ assigning the obtained values to the corresponding cells states.

Application of $\theta(i,j)$ to all $(i,j) \in M)$ transforms a cellular array $\Omega(t)$ into the next-state one $\Omega(t+1)$. The sequence

$$\Sigma(\Omega) = (\Omega(0), \Omega(1), ..., \Omega(t), \Omega(t+1), ..., \Omega(\hat{t})), \tag{8}$$

obtained during iterative operation on the CA is called *the evolution*, t being the iteration number, and $\Omega(0)$ - the initial cellular array. During the evolution the cellular array changes its “black-white” pattern tending to a stable one.

Each CA produces a scope of patterns characterized by similar features, which define a certain motif. There is no strict method for CA synthesis by the given properties of the resulting pattern. Only a weak correspondence between the transition function and produced motifs may be known. Moreover, in the range of one and the same CA and the corresponding motif, a great variety of patterns, differing in tortuosity, porosity and orientation of black-white borders may be obtained by variation of CA initial cellular array. Also, when running the iterative process of evolution it is possible to observe the sequence of produced patterns and stop the process at any moment when $\Omega(t)$ meets the wanted parameters.

Example 1. The CA $\aleph_{P1} = \langle A_1, M_1, \theta_1 \rangle$ is a model of the process of phase-separation. Being applied to a cellular array $\Omega(0)$ with randomly distributed "ones" and "zeroes", this CA gradually aggregates the "ones" in patches of fancy forms. The parameters of $\aleph_{P1}$ are as follows. $A_1 = \{0, 1\}$, the size is $I \times J = 300 \times 300$,

$$\theta_1 : \{(v_{kl}, (i-k, j-l)) : k, l = -r, ..., r\} \to (v', (i,j)), \tag{9}$$

where

$$v' = \begin{cases} 1, & \text{if} \quad s = (q-1)/2 \quad \text{or} \quad s > (q+1)/2, \\ 0, & \text{if} \quad s = (q+1)/2 \quad \text{or} \quad s < (q-1)/2, \end{cases} \tag{10}$$

where

$$q = (2r+1)^2, \quad s = \sum_{kl=0}^{q} v_{kl}$$

In Fig.1 three patterns produced by $\aleph_{P1}$ are shown. The simulation was performed with periodic boundary conditions. The initial cellular array is a random distribution of "ones" over M_1 with average density $\rho = 0.5$. From the resulting evolution of $\aleph_{P1}$ the properties of porous media represented by the obtained pattern, are easily computed according to (1-3). Thus, the patterns in Fig.2 are characterized by the porosity $Por \simeq 0.6$. The tortuosity of the pattern at $t = 10$ is 10 times larger than that of $t = 50$. As for the resistance, it is equal for both patterns, being approximately $Res \simeq 0.55$.

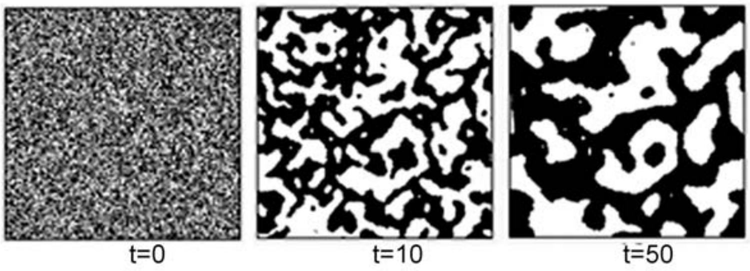


Fig. 1. Three snapshots obtained while simulating the evolution of a CA $\aleph_{P1}$, given by (9,10)

Example 2. Two CAs $\aleph_{P2} = \langle A_2, M_2, \theta_2 \rangle$, and $\aleph_{P3} = \langle A_3, M_3, \theta_3 \rangle$, which differ in mode of operation and in initial cellular arrays produce the patterns with different motifs.

Both CAs have Boolean alphabets, are of equal size $I \times J = 400 \times 400$. The first one ($\aleph_{P2}$) operates in synchronous mode. Being applied to a cellular array $\Omega_2(0)$ with randomly distributed "ones" having the density $\rho = 0.001$, it produces "round spots" of equal size randomly located over the array. In the process of the evolution the spots grow in diameter, exhibiting the decrease of the porosity coefficient of the corresponding cellular array. When the porosity match the wanted value the evolution stops.

The local operator is as follows.

$$\theta_2 : (v_0, (i,j)), (v_1, \phi_1(i,j)), \dots, (v_q, \phi_q(i,j))\} \to (v_0', (i,j)), \qquad (11)$$

where $q = (2r+1)^2$, $r = 3$ being the radius of neighborhood template, $q = 49$.

$$\phi_l(i,j) = (i + g_l, j + h_l), \quad g_l = l_{mod(2r+1)} - r, \quad h_l = \lfloor l/(2r+1) \rfloor;$$

$$v_0' = \begin{cases} 1, & \text{if} \quad \sum_{l=0}^{q} w_l v_l > 0 \\ 0, & \text{otherwise,} \end{cases} \qquad (12)$$

where

$$w_l = \begin{cases} 1, & \text{if} \quad g_l \leq 1 \;\&\; h_l \leq 1 \\ -0,2 & \text{otherwise.} \end{cases}$$

Sum in (12) may be also obtained by imposing a weighted template

$$\mathrm{W} = \begin{array}{|c|c|c|c|c|c|c|} \hline a & a & a & a & a & a & a \\ \hline a & a & a & a & a & a & a \\ \hline a & a & 1 & 1 & 1 & a & a \\ \hline a & a & 1 & \mathbf{1} & 1 & a & a \\ \hline a & a & 1 & 1 & 1 & a & a \\ \hline a & a & a & a & a & a & a \\ \hline a & a & a & a & a & a & a \\ \hline \end{array} \quad , \qquad a = \text{-0.2},$$

onto a cell and computing the sum of products of its entries by underlying cell states. A snapshot at $t = 10$ of the evolution of $\aleph_{P2}$ is shown In Fig.2a.

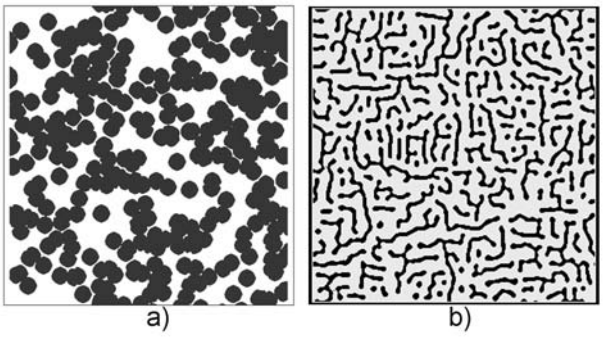


Fig. 2. Two patterns obtained by evolution of two CAs with weighted templates, and θ_2, given by (11,12): **a**) with synchronous mode of operation and initial density $\rho = 0,001$, b) with asynchronous mode of operation and initial density distributions $\rho_i = 0.008$

$\aleph_{P3}$ operates in asynchronous mode, starting with an initial cellular array $\Omega_3(0)$ having “ones” and “zeroes” distributed randomly with the density $\rho_i = 0.008$. The obtained pattern is inverted turning the formatted black stripes into solids and leaving the the white area for pores. The obtained pattern is a stable one (Fig.2b).

The two above examples are given just to show that a great variety of porous structure may be obtained in the form of a Boolean array by using pattern formation CA. Among them are the widely used in porous medium investigation circles (Fig.2a) and fibers (Fig2.b)

4 CA-Model of the Flow

The CA-model for fluid flow simulating, referred further as the FlowCA and denoted as $\aleph_F = \langle A_f, M_F, \Theta \rangle$, is a Gas-Lattice FHP CA-model [4,5] with special boundary conditions. The flow is represented by abstract particles, moving and colliding in a discrete hexagonal space. The naming set is given by the set of hexagons coordinates. In our implementation we use the most simple way of mapping the hexagonal space onto the rectangular lattice, which implies the cell centers being allocated as a chessboard assuming each cell to occupy a pair of coordinates along one (say the jth) of the axes, thus having the six neighbors (Fig.3 a). The distances between two neighboring cells are assumed to be equal to 1. So, $M = \{(i,j) : i = 0,1,\ldots,I; j = 0,1,\ldots,J\}$. The naming template

$$T(i,j) = \{(i,j), \phi_1(i,j), \ldots, \phi_6(i,j)\}, \tag{13}$$

where

$$\begin{array}{lll} \phi_0(i,j) = (i,j), & \phi_1(i,j) = (i,j+2), & \phi_2(i,j) = (i-1,j+1), \\ \phi_3(i,j) = (i-1,j-1), & \phi_4(i,j) = (i,j-2), & \phi_5(i,j) = (i+1,j-1), \\ & \phi_6(i,j) = (i+1,j+1)\ . & \end{array} \tag{14}$$

Each cell may have up to 6 particles, each being provided with a velocity vector directed towards one of the neighbors. The cell state alphabet represents the 6 moving particles by Boolean vectors 6 bit long: $A = \{(v_1, \ldots, v_k, \ldots, v_6) : v_k \in \{0,1\}, |A| = 2^6$. A component $v_k = 1$ of a state vector indicates that the cell $(\mathbf{v}, (i,j))$, has a particle moving towards the kth neighbor with a velocity $v_k = 1$. Particle mass is equal to 1.

The set of two local operators $\Theta = \{\theta_1, \theta_2\}$ determines the CA functioning, θ_1 representing a propagation substep, θ_2 – simulating the collision.

$$\theta_1(i,j) : \{(\mathbf{v_0}, (i,j)) \ldots, (\mathbf{v_l}, \phi_l(i,j)), \ldots, (\mathbf{v_6}, \phi_6(i,j))\} \to \{(\mathbf{v_0\prime}, (i,j))\}, \tag{15}$$

where

$$\mathbf{v\prime}(i,j) = \bigvee_{l=0}^{6} v_l(\phi_{l+3}(i,j)). \tag{16}$$

Here and further summation of indices is $+_{mod6}$.

The collision operator is as follows.

$$\theta_2(i,j) = \{(\mathbf{v_0}, (i,j))\} \rightarrow \{(\mathbf{v_0\prime}, (i,j))\}. \tag{17}$$

The transition function in (17) $\mathbf{v_0\prime} = f(\mathbf{v_0})$ is given in the form of a table, some arguments having two equiprobable outcomes. The principles of collision rules functioning is shown in Fig. 3b. The essential requirement is the satisfaction of the lows of mass and momentum conservation. Of course, all rules obtained by rotation those given in Fig.3b are also included in the transition rules set.

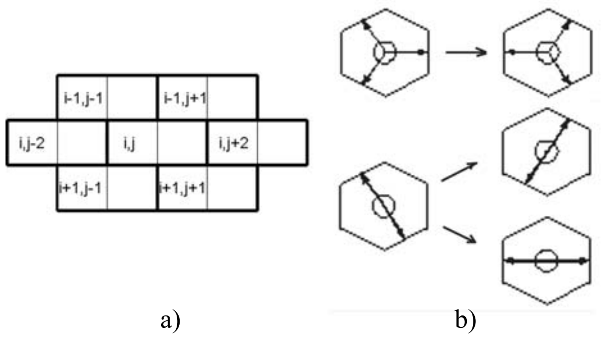


Fig. 3. Graphical representation of a) hexagonal Lattice mapping onto the rectangular one, and b) collision operators in FHP model

The mode of operation of Lattice-Gas CA is two-stage synchronous, i.e. each iteration consists of two stages: the propagation operators (16) act on the first stage, on the second stage the collision operator completes the transition to the next global state, i.e. $\Theta = \theta_2(\theta_1)$. The set of cells is partitioned into four parts: pore cells Ω_P, wall cells Ω_W, input cells Ω_{In}, and output cells Ω_{Out}, the collision rules ifor which being different.

For representing the interactions between the pore cells and walls the collision rules are used, referred to as *border rules.* They depend on tension forces between the liquid and the solid pore wall surface, which determines the wettability of porous material. So, the wall-cell collision rules are of two types: 1)the *no-slip* rules, which are, sometimes, referred to as *bounce-back* boundary conditions, used in hydrophobic porous materials, and 2) the slip rules expressing *slip* boundary conditions, which characterize the hydrophilous pore materials. In both cases the border rules are applied to the cells belonging to a layer of $B \in \Omega_W$ alongside the pores, referred to as a *border layer*, i.e. $(i,j) \in B$ if $T(i,j) \bigcap \Omega_P \neq \emptyset$.

The no-slip collision rule prescribes the particle entering the border layer cell to reverse its direction. Naturally, the rule does not conserve the momentum, due to the additional tension forces between solid and liquid. As for mass conservation law, it is obviously satisfied with any pore tortuosity. So, the collision rule in border layer cells for no-slip border condition is as follows

$$\theta_{n-s} : (\mathbf{v}, (i, j)) \to (\mathbf{v\prime}, (i.j)) \quad \forall (i, j) \in B, \tag{18}$$

where the components of $\mathbf{v\prime}(i, j)$ are

$$v'_k = v_{k+3} \tag{19}$$

The above no-slip collision rule provides fluid velocity to be zero in the direct vicinity of the walls. If more strong hydrophobicity is wanted, then some fictitious wall cells should be added along the border which increase roughness of the border. Those cells have to replace the pore cells, $(i, j) \in \Omega_P$ and should meet the condition $|T(i, j) \bigcap \Omega_P| \geq 4$. They are chosen randomly with probability ranging from $p = 0$ to $p = 0.5$, maximum being reached with $p = 0.5$.

The slip collision rule prescribes a particle entering the border layer to continue moving along the channel wall changing the direction as less as possible. In porous media with regular straight channels this principle is realized by *specular reflection* rules. In tortuous porous medium with arbitrary border curvature this rule should be transformed into a number of rules depending on mutual location of the border cell relative to its neighbors in the wall and on the number of wall cells in the cell neighborhood. Hence, the collision operator being the same as in no-slip case (18) has the following transition functions.

If $T(i, j) \bigcap \Omega_P = \{\phi_k(i, j), \phi_{k+1}(i, j)\}$, then

$$v'_k = v_{k+4}, \quad v'_{k+1} = v_{k+3}, \quad k = 1, \ldots, 6, \tag{20}$$

otherwise the transition function should be the same that in no-slip case (19).

The above slip collision rules do not provide strong hydrophility. When stronger hydrophility is wanted, then a procedure similar to that for increasing hydrophobity should be applied. Namely, some fictitious wall cells adjacent to the border ones, should be added in such a way that smooth the border. This condition yields in $|T(i, j) \bigcap \Omega_P| \leq 2$. The cells meeting this condition are chosen randomly with probability ranging from $p = 0$ to $p = 0.5$, maximum being reached with $p = 0.5$.

The proposed approach is further illustrated by an Example of simulating the flow of water through a carbon hydrophobic cathode in Proton Exchange Fuel Cell [1]. But beforehand, in order to test the presented approach in more details, a simulation is performed on a small prototype fragment.

5 Implementation of the Method

The carbon porous specimen and its small prototype fragment are shown in Fig.4. The averaged diameter of pore channels in the material under investigation is about 10μ.

Following the method of mapping physics onto CA-models from[15], it is possible to obtain all parameters of the model. Since, according to the theory and

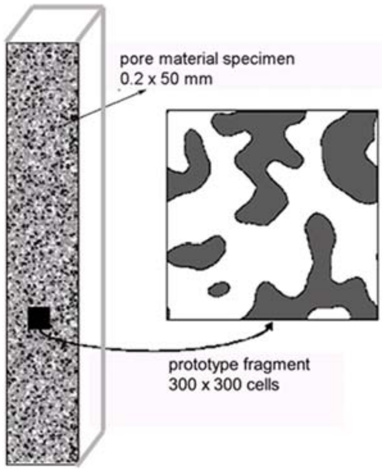


Fig. 4. A porous carbon cathode and its small prototype fragment, used for the computational test experiments

practice of porous medium [2] simulation, the smallest channel should be $\simeq 10$ times wider than model length units, the length scale is chosen to be $h = 0.2 \simeq \mu$, which yields the model size of the given specimen to be 1000×250000 cells. The viscosity scale $Sc(\nu)$ is also straightforward, because the viscosity of FHP-model is proved to be $\nu_{CA} = 0.85$ [5], and the viscosity of water is $\nu_w = 10^{-4}\ \mathtt{m^2s^{-1}}$, yielding $Sc(\nu) = 1.17 \cdot 10^{-5}\ \mathtt{m^2s^{-1}}$. These two scales allow to calculate all others relations between the CA-model and their physical counterparts. They are as follows:

particle mass $m_p = (\rho_w/(h^3 \times 6) = 2.7 \cdot 10^{-18}$ kg,
time step $\tau = h^2/Sc(\nu) = 3.5 \cdot 10^{-8}$ s,
velocity scale $Sc(v) = h/\tau = 0.57 \cdot 10^{-3}$ ms^{-1},
flux scale $Sc_f = F \cdot m_p/\tau$ kgs^{-1}, F defining the number of particles passing through the fragment per iteration, computed during the simulation process.

The last scale is the most important because it is a main characteristic of a porous materials.

For testing the proposed method on a personal computer and to observe the process in its dynamics a prototype fragment of the size 300×300 cells has been chosen, which is 2800 times less than the real length. The simulation has been performed for a number of different fragments, obtained by changing the parameters of the pattern formation CA. The velocity fields and the flows values have been obtained. In Fig. 5 on the left side a velocity field of a flow through a hydrophobic prototype fragment is shown, obtained after $t = 4000$ iterations when the process reached the stable state. In Fig.5 on the right side four profiles of density and averaged velocity, taken in the vertical cut of the middle of the

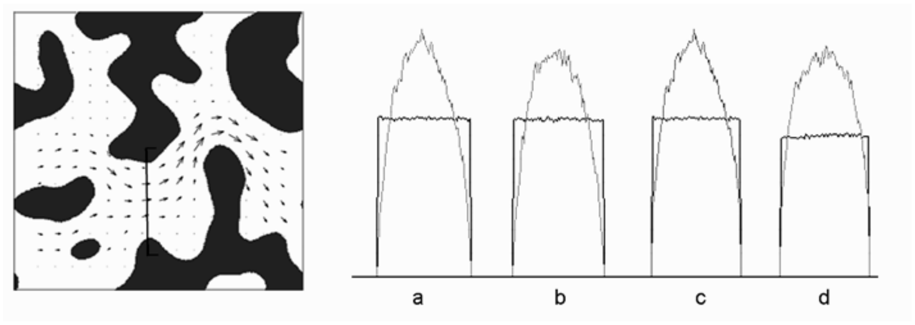


Fig. 5. The snapshots of fluid flow simulation through the hydrophobic prototype fragment: on the left side the velocity field is shown, on the right side the profiles of density (the black curve) and the velocity (the gray curve) through the vertical cut in the narrow part of the channel, marked by a dark line

Table 1. Flow Q through the fragment (Fig.5) and mean velocities $\langle v \rangle$ in its vertical cut at $t = 4000$ for four cases shown in Fig.5

case	property	Q	$\langle v \rangle$
a	hydrophobic	39.5	0.88
b	strong hydrophobic	36.7	0.85
c	weak hydrophilous	40.9	0.86
d	hydrophilous	42.1	0.87

fragment are shown for different border conditions of the pore walls. In Table 1 the averaged velocities in the cut (in units per iteration), and flows through the fragment (in particle numbers per iteration) (F/t), are given for the four cases, shown in Fig.5.

For investigating the real size specimen (0.2×50mm) the simulation has been performed on 128 computers (the Cluster K-100 of Joint Supercomputer Center of Russian Academy of Sciences), using MPI library. The cellular array was decomposed into 128 domains each being of size 1000×2000. For constructing the 2D representation of the porous medium a “phase separation” CA with $R = 3$ (Section 3) was used. The total flow F, obtained for hydrophobic specimen with porosity Por=0,608 is 83867 particles/iteration, which makes 0.319 kg/s through a 3D specimen of one cm wide. The efficiency of parallelization in an eight-core node is computed as

$$E_{node} = T(1) \cdot 8/T(8) \simeq 0.2,$$

where $T(1)$ is the time of one domain processing in a single core, $T(8)$ is the time of eight domain processing on in an 8-core node. The efficiency of parallelization in a cluster

$$E_{cluster} = T(8) \cdot 16/T(128) \simeq 0.982,$$

where $T(128)$ is the time of processing 128 domains in 16 eight-core nodes.

6 Conclusion

A method for simulation fluid flow at pore level though porous materials with different wettability of pore walls is presented. The method combines the construction of the porous material structure and simulation the flow of gas or liquid through it. Both stages are based on classical CA simulation methods, which manifests their capability and the extent of CA applications. The implementation of the method in a single computer as well as in a cluster showed its simplicity of programming and efficiency of parallel implementation

References

1. Larminie, J., Dicks, A.: Fuel Cells Systems Explained. John Wiley & Sons, New York (2003)
2. Sahimi, M.: Flow phenomena in rocks: from continuum models to fractals, percolation, cellular automata and simulated annealing. Rev. Modern Physics 65(4), 1393–1533 (1993)
3. Garboczi, E.J., Bentz, D.P., Snyder, K.A., Martys, N.S., Stutzman, P.E., Ferraris, C.F., Jeffrey, W.: Modeling And Measuring the Structure And Properties of Cement-Based Materials (An electronic monograph), `http://ciks.cbt.nist.gov/garbocz/appendix2/node8.html`
4. Rothman, B.H., Zaleski, S.: Lattice-Gas Cellular Automata. Simple Models of Complex Hydrodynamics. Cambridge Univ. Press, London (1997)
5. Frish, U., d'Humieres, D., Hasslacher, B., Lallemand, P., Pomeau, Y., Rivet, J.P.: Lattice-Gas hydrodynamics in two and three dimensions. Complex Systems 1, 649–707 (1987)
6. Succi, S.: The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. Oxford University Press, New York (2001)
7. Nabovati, A., Sousa, A.C.M.: Fluid Flow Simulation In Random Porous Media At Pore Level Using The Lattice Boltzmann Method. J. of Eng. Sci. and Techn. 2(3), 226–237 (2007)
8. Clague, D.S., Kandhai, D., Zang, R., Sloot, P.M.A.: Hydraulic permeabolity of (un)bounded fibrous media using the Lattice Boltzmann method. Physical Review E 61(1), 616–625 (2000)
9. Pan, C., Hilpert, M., Miller, C.T.: Pore-scakle modeling of saturated permeabilities in random sphrere packings. Physical Review E 64(6), Article N 006702 (2001)
10. Bandman, O.: Composing Fine-Grained Parallel Algorithms for Spatial dynamics Simulation. In: Malyshkin, V.E. (ed.) PaCT 2005. LNCS, vol. 3606, pp. 99–113. Springer, Heidelberg (2005)
11. CNN: a Paradigm for Complexity. World Scientific, Singapore (2002)
12. Wolfram, S.: A new kind of science. Wolfram Media Inc., Champaign (2002)
13. Toffolli, T., Margolus, N.: Cellular Automata Machines. MIT Press, USA (1987)
14. Achasova, S., Bandman, O., Markova, V., Piskunov, S.: Parallel Substitution Algorithm. Theory and Application. World Scientific, Singapore (1994)
15. Bandman, O.: Mapping physical phenomena onto CA-models. In: Adamatsky, A., Alonso-Sanz, R., Lawiczak, A., Martinez, G.J., Morita, K., Worsch, T. (eds.) AUTOMATA 2008. Theory and Application of Cellular Automata, pp. 391–397. Luniver Press, UK (2008)

Solving All-to-All Communication with CA Agents More Effectively with Flags

Patrick Ediger and Rolf Hoffmann

Technische Universität Darmstadt
FB Informatik, FG Rechnerarchitektur
Hochschulstr. 10, 64289 Darmstadt, Germany
{ediger,hoffmann}@ra.informatik.tu-darmstadt.de

Abstract. We have investigated the all-to-all communication problem for a multi-agent system modeled in cellular automata. The agents' task is to solve the problem by communicating their initially mutually exclusive information to all the other agents. In order to evolve the best behavior of agents with a uniform rule we used a set of 20 initial configurations, 10 with border, 10 with cyclic wrap-around. The behavior was evolved by a genetic algorithm for agents with (1) simple moving abilities, (2) for agents with more sophisticated moving abilities and (3) for agents with indirect communication capabilities (reading and writing flags into the environmental cells). The results show that the more sophisticated agents are not only more effective but also more efficient regarding the effort that has to be made finding a feasible behavior with the genetic algorithm.

Keywords: cellular automata, multi-agent system, evolving behavior, different action sets.

1 Introduction

The general goal of our project is to develop methods to optimize the local behavior of moving agents in a multi-agent system in order to fulfill a given global task. In this investigation we particularly concentrate on the moving and communication abilities of the agents in order to find out whether increasing the complexity of an agent can lead to a better behavior without increasing the effort for the optimizing procedure at the same time.

The global task we chose for this investigation is the all-to-all communication task: Several agents are moving around in a cellular automata (CA) grid. Each one initially has got one part of the mutually distributed information which can be exchanged when the agents meet in certain defined local patterns (communication situations). The task is considered *successful* when all agents have gathered the complete information. Possible communication situations are shown in Fig. 1. In the cases a, b, c the agents are directly in contact. But it is a matter of definition whether such situations allow communication. For this investigation

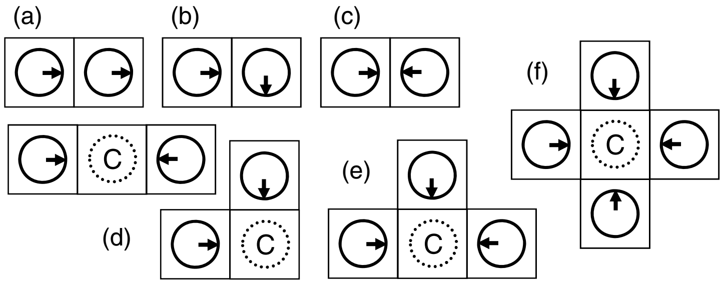


Fig. 1. Communication situations. Communication is only allowed for the cases d, e, f using a mediator C.

we have defined the patterns d, e, f to be the only ones which allow communication. A reason could be that communication can only take place if a mediator/negotiator is used between them. Furthermore the mediator may perform a particular computation (e. g., average, maximum, priority select). Such conflicts occur when agents want to move to the same target position, like vehicles which are meeting in a cross-way. The center of the crossing can be interpreted as the mediator.

In this contribution we pose the following questions: Which abilities do the agents need to efficiently solve the given problem? Regarding that adding more abilities to the agents raises their complexity, does the optimization become harder or can we evolve better agent behaviors with the same amount of optimization time?

In former investigations [1] 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 presented problem is related to it with respect to finding an optimal movement of the agents. But the task is different: Now the agents shall exchange their information in shortest time taking advantage out of the conflicts which are useful and necessary for the all-to-all communication.

All-to-all communication is a very common task in distributed computing. 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. All-to-all communication in multi-agent systems is related to multi-agent problems like finding a consensus [2], synchronizing oscillators, flocking theory or rendezvous in space [3], or in general to distributed algorithms with robots [4]. We have already studied the problem of all-to-all communication [5,6]. In both investigations a grid of size 33×33 without obstacles and 16 randomly distributed agents at the beginning were used. There were three types of environments given in which the task had to be fulfilled: An environment with border EnvB, an environment without border EnvC (cyclic, wrap-around) and a *"dual environment"* EnvBC.

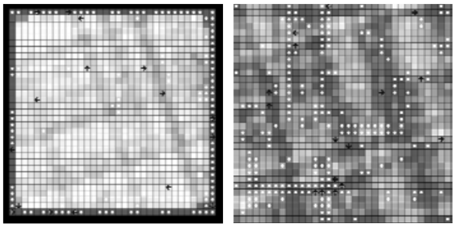


Fig. 2. Patterns emerged by simulation for the best algorithm for a particular initial configuration evolved for the *dual* EnvBC (applied to EnvB with border and also applied to EnvC with wrap around). The white spots indicate positions of mediators where an information exchange has occurred.

The dual environment joins EnvB and EnvC meaning that the agents shall be able to communicate successfully in both environments. We will restrict this investigation to the dual environment.

The best algorithms evolved for EnvBC [5] needed 773.2 steps (CA generations) averaged over 50 random initial settings of the agents with border and 50 without border. Fig. 2 shows the typical patterns caused by the visited cells of the agents (the darker the more often visited). It should be noted that the algorithms were evolved before for the communication situations a, b, c instead of d, e, f now (Fig. 1) and that the agents used a very limited action set (if cannot move forward: turn right/left, if can move forward: turn right/left and move forward simultaneously). This limited action set (in section 2 denoted as RX/LX) will later be used for comparison (task (1) in section 4) but the goal is to evolve agents with a more powerful action set.

Our research in general is also related to works like: evolving optimal rules for cellular automata (CA) [7,8], finding out the center of gravity by marching pixels by evolutionary methods [9], modeling multi-agent systems in CA to simulate pedestrian movement [10] or traffic flow [11].

The remainder of this paper is structured as follows. In Section 2 the modeling of the multi-agent system (MAS) including the possible action sets for the agents are explained. The genetic procedure used to evolve the agents' behavior is described in Section 3. Section 4 provides the results of this investigation and Section 5 concludes.

2 CA Modeling of the Multi-agent System

The whole system is modeled by cellular automata. It consists of an environment ($n \times m$ grid) with borders or without borders (wrap-around) and k uniform agents. An agent has a certain moving direction and it can only read the information from one cell ahead (target cell, front cell). If it detects a border cell or an agent in front or a conflict, it will stay on the current cell. A conflict occurs when two or more agents want to move to the same front cell (crossing point, cell in conflict, mediator). In order to detect a conflict an extended neighborhood [5] is needed (Manhattan distance of 2 in the moving direction). Alternatively the conflict detection can be realized by an arbitration logic [1] which is 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 model the distribution of information we are using a bit vector with k bits which is stored in each agent. At the beginning the bits are set mutually exclusive (bit(i)=1 for agent(i)). When two, three or four agents form a communication situation they exchange their information by simply OR-ing their bit vectors together. The task is successfully solved when the bit vectors of all agents obtain $11 \ldots 1$.

In the case that the agent can move forward it will move forward. In addition to the movement ahead, which is not decided by the agent but implicitly by its local environment, an agent decides to perform simultaneously a turning action:

– R: turn 90 degrees to the right.
– L: turn 90 degrees to the left.
– S: stay (or "straight") in the same direction as before.

Apart from the agent's movement and the information exchange, an agent has indirect communication capabilities. Each cell of the environment contains a (status) flag f which is either 0 or 1 and used as an input for the decision making process. The flag's status 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 perform three different actions on the flag of the cell on which the agent is currently located:

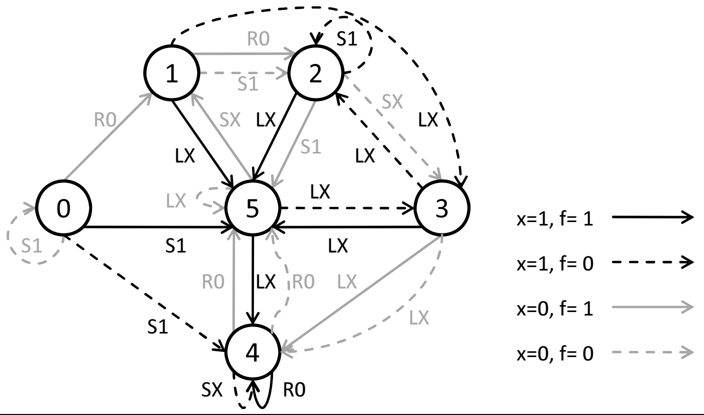
– *0*: set flag to value 0.
– *1*: set flag to value 1.
– X: leave flag value as it is.

Thus in total there are nine possibilities of actions that an agent can perform in one generation: $R0$, $R1$, RX, $L0$, $L1$, LX, $S0$, $S1$ and SX. The agent performs the rule:

1. (*Evaluate move condition* x): If (front cell == OBSTACLE $\vee$ AGENT $\vee$ CONFLICT) then $x = 0$ else $x = 1$
2. (*React*): If (x) then move forward *and* perform action $R0 \ldots SX$, else perform action $R0 \ldots SX$

The decision which of the actions will be performed depends on the behavior of the agent. The behavior (algorithm) of an agent is defined by a finite state machine. Input of the state machine is the move condition x and the status flag f. Output of the state machine is the signal y whose values are mapped to the possible actions described before (see also Table 1).

A state machine is defined by a state transition table (Fig. 3) with current input x and f, current state s, next state s' and current output y. In order to



f	0												1											
x	0						1						0						1					
s	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
s'	0	2	3	4	5	5	4	3	2	2	4	3	1	2	5	4	5	1	5	5	5	5	4	4
y	*S1*	*S1*	*SX*	*LX*	*R0*	*LX*	*S1*	*LX*	*S1*	*LX*	*SX*	*LX*	*R0*	*R0*	*S1*	*LX*	*R0*	*SX*	*S1*	*LX*	*LX*	*LX*	*R0*	*LX*
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Fig. 3. A state table and the corresponding state graph, defining the behavior (algorithm) of an agent, restricted to 6 states and the actions $R0$, LX, $S1$ and SX

keep the control automaton simple, we restrict the number of states and actions to a certain limit (see Sec. 3). Modeling the behavior with a state machine with a restricted number of states and evaluation by enumerations was also undertaken in SOS [12].

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 33 × 33. This size was taken over from former investigations allowing to distribute equally a varying number of agents at the borders. In this investigation the number of agents is set to $k = 16$. From former investigations in multi-agent systems we know that a number of agents between approx. 8 and 64 can lead to good synergy effects and a sufficient number of conflicts which are required here.

3 The Genetic Procedure

The ultimate goal is to find the optimal behavior on average for all possible initial configurations. As we cannot test all possible configurations we will be satisfied if we can find the best behaviors for a test set of 20 randomly generated initial configurations (10 with border and 10 with wrap-around), i. e., randomly generated starting positions and directions for the 16 agents. 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 statetable is $K = (\#s\#y)^{(\#s\#x\#f)}$ where $\#s$ is the number of states, $\#x\#f$ is the number of different input values and $\#y$ is the number of different output actions. Therefore we used a genetic procedure and tried to find the best behavior within a reasonable computational time limit. In the former investigations [5] we have experimented with $\#x = 2$ (only move condition, no flags), $\#y = 2$ (allowing only the actions RX and LX) and $\#s = 6$ in order to keep the control automaton as simple as possible. With these values 12^{12} possible state tables can be defined. Note that not all of these represent distinct behaviors (e. g., permutation of the states leads to equivalent behaviors) or useful behaviors (e. g., state graphs which make little use of the inputs or which are weakly connected). If we add the flag status information to the inputs ($\#x = 4$) and for example two more actions ($\#y = 4$), our formula gives us in total 24^{24} possible state tables, meaning that the search space increases exponentially.

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 (start positions and direction of the creatures) under test. In other words we search for state algorithms which can solve the problem with a minimum number of steps.

A concatenation of the pairs (s', y) in the state table (Fig. 3) is a string representation and defines the genome of one individual, a possible solution. P populations of N individuals are updated in each generation (optimization iteration). During each iteration M offsprings are produced in each population. The union of the current N individuals and the M offsprings are sorted according to their fitness and the N best are selected to form the next population. An offspring is produced as follows:

1. (Get Parents) Two parents are chosen for each population. Each parent is chosen from the own population with a probability of p_1 and from an arbitrary other population with the probability of $(1 - p_1)$.
2. (Crossover) Each new component (s'_i, y_i) of the genome string is taken from either the first parent or the second parent with a probability of 50%. This means that the tuple (next state, output) for the position i=(input, state) is inherited from any parent.
3. (Mutation) Each component (s'_i, y_i) is afterwards mutated with a probability pf p_2. Thereby the next state and the output at position i are changed to a random value.

The fitness function F is evaluated by simulating the agent system. It reflects three aspects:

1. The number of agents (maximum 16) 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 algorithm should perform at least one communication.

3. The number of steps in the CA simulation to reach successfulness. We will call this value also *communication time*.

The used fitness function integrates these aspects by using appropriate weights:

$$F = 10^5(16 - a) + 10^4(1 - c) + t_c$$

where a is the number of informed agents, $c = 1$ if at least one communication took place, $c = 0$ if not, and t_c is the communication time.

The first and second part of the fitness function are only relevant at the beginning of the genetic algorithm (GA) for the ordering of the poor individuals. After the genetic algorithm has proceeded a certain time only successful algorithms are held in the populations. For successful algorithms the relation *Fitness = communication time = steps* holds. Note that a lower fitness value is better.

4 Results

We evaluated the *effectiveness* and *efficiency* for the indirect communication through flags and for the different moving abilities. As indicator for effectiveness the fitness (quality) of the evolved algorithms is used, while as indicator for efficiency the computational effort of the genetic algorithm is used. For that purpose we carried out the following five tasks (Fig. 4):

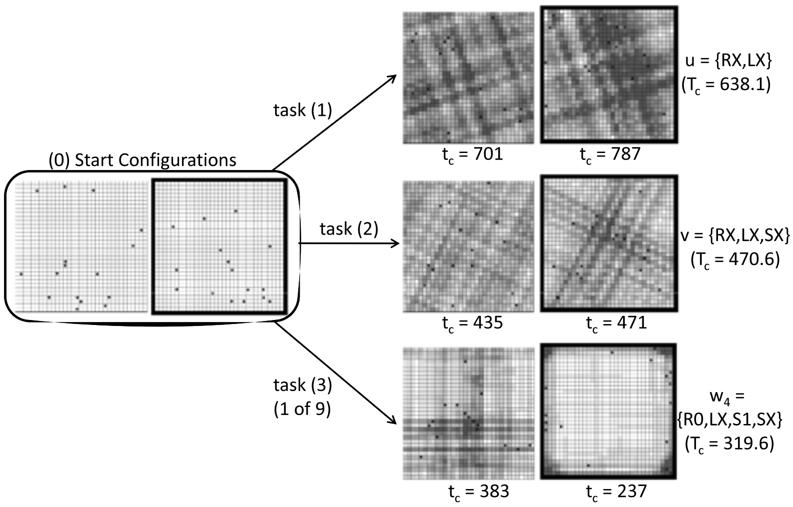


Fig. 4. Schematic depiction of the different tasks carried out. Right hand side shows the typical patterns emerged by the best algorithm found for each of the action sets $\{RX, LX\}$, $\{RX, LX, SX\}$ and $\{R0, LX, S1, SX\}$. T_c indicates the average of the communication times t_c over all 20 environments.

Table 1. Mapping of the output y to the corresponding actions of the nine selected action sets w_i, u and v

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	u	v
$y=0$	$R0$	$R0$	$R0$	$R0$	$R0$	$R1$	$R1$	$R1$	RX	RX	RX
$y=1$	$L0$	$L1$	$L1$	LX	LX	$L1$	LX	LX	LX	LX	LX
$y=2$	$S1$	$S0$	$S1$	$S0$	$S1$	$S0$	$S0$	$S0$	$S0$		SX
$y=3$	SX	SX	SX	$S1$	SX	SX	$S1$	SX	$S1$		

(0) (Initialize) 10 random initial configurations for the environment with border (EnvB) and 10 without (cyclic, EnvC) were computed in advance and used for all experiments.
(1) (Simple Movement) The genetic procedure was performed evolving algorithms U with $s = 6$ states restricted to the actions $u = \{RX, LX\}$, i. e., without the ability to walk straight or to read and write flags.
(2) (Complex Movement) The genetic procedure was performed evolving algorithms V with $s = 6$ states restricted to the actions $v = \{RX, LX, SX\}$, i. e., with the additional ability to walk straight but not to read and write flags.
(3) (Tracing Agents) The genetic procedure was performed evolving algorithms W_0 - W_8 with $s = 6$ states and nine different sets (w_0 - w_8, Table 1) of $\#y = 4$ different output actions, including the ability to walk straight and to read and write flags.
(4) (Evaluation) The evolved algorithms U, V and W_0 - W_8 were compared.

We used $\#s = 6$ possible states because we yielded good results for this amount in former investigations. In order to limit the search space to a reasonable number we restricted the possible outputs in task (3) to $\#y = 4$. This also gives us the chance to compare among the algorithms W_i by using different action sets with the same complexity. There are in total 126 possible combinations taking four out of nine actions. We reduced this number to nine (Table 1) by the following constraints:

1. At least one of each action changing the flag status ($0/1/X$) should be included (45 combinations left).
2. Demanding symmetry and the ability to walk straight, the four moving actions should consist of one R-, one L- and two S-actions (15 combinations left).
3. If interchanging R and L leads to another action set among the 15, then keep only one of the two (9 combinations possible).

The algorithms were evolved using $P = 3$ populations with $N = 100$ individuals each, $M = 10$ offsprings, $p_1 = 2\%$ and $p_2 = 9\%$. For each task six independent runs were performed for $g = 10{,}000$ generations. The results were averaged over these six runs whenever it makes sense.

The first task delivered one algorithm U that could solve all 20 environments on average in $T_c = 638.1$ steps. Taking into account the 10 best algorithms of each run of the GA and averaging over the six runs, the mean value is 661.9 steps. The best algorithm V however needs only $T_c = 470.6$ steps, while the average value of the top 10 over all six runs is 542.3. This means that walking straight (SX) is an effective action and should be considered when designing rules for the agents. The fact that in both cases 10,000 generations were used in all runs means that the increasing complexity of the control automata does not seem to make it harder for the GA to find a feasible solution.

These results were also compared to random walkers. 1,000,000 runs were simulated with the moving abilities RX, LX and SX. On average the 16 random walkers needed 1,038 steps to communicate successfully. In all runs the communication was completely successful while the communication time was distributed between 825 and 1,290. Random walkers with the actions RX and LX only performed even worse needing 1,311 steps on average. Compared to the best algorithms U and V the random walkers are significantly slower.

The values T_c averaged over the ten best found algorithms W_i and over the six runs range from 353.1 to 418.6 (Fig. 5), which is considerably better than the evolved algorithms without the ability to communicate through flags. The best algorithm that was found is shown in Fig. 3. It uses the actions set $w_4 = \{R0, LX, S1, SX\}$ and needs $T_c = 319.6$ steps to complete the task. Obviously the ability to use flags is effective.

Despite the increasing complexity of the algorithms ($U < V < W$), the GA is able to find better solutions W_i than V with the same amount of generations. But apart from that observation, we also compared the actual computation time for the GA runs. It turned out that the GA evolving algorithms W_i needed 14.1 hours on average on a PC (dual core 2GHz) for six runs. The GA to evolve algorithms V needed 15.6 hours while evolving algorithms U took 18.3 hours of computation time on the same machine. This is due to the fact that the fitness correlates with the simulation time and fitness was obviously better in the case of task (3). Adding the new actions (move straight, use flags) to the agents' behavior thus is effective and efficient.

In Fig. 5 the algorithms W_i are sorted by communication time (averaged Top 10 evolved algorithms). The three worst evolved algorithms are W_1, W_7 and W_5, which all comprise the actions $S0$ and SX. It seems the combination of these two actions is not so effective. In order to find out the reason, we further investigated which of the four actions the agents actually use during simulation. Therefore we analyzed the data of all the top 10 algorithms of all runs of all action sets w_i. It turned out that in all nine cases the agents preferred to take the actions $y = 2/3$ (stay or walk straight) instead of $y = 0/1$ (turn left or right). Averaged over all nine combinations the actions $y = 2/3$ were performed in 72.8% of all steps, although there is no significant overweight of these outputs in the state tables. According to that, walking straight is a good strategy for

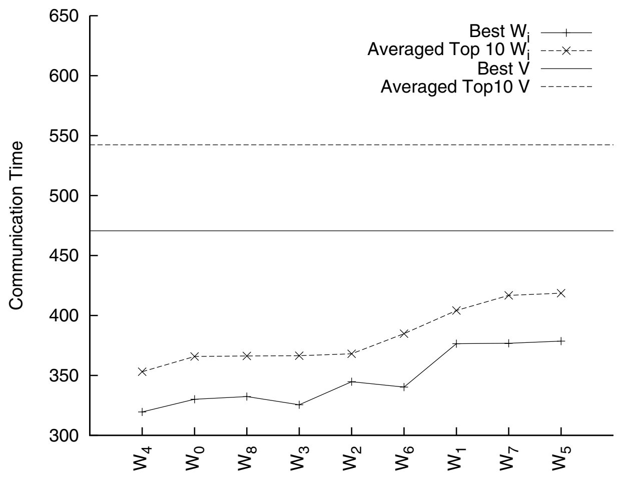


Fig. 5. The best algorithms W_i and the averaged top 10 evolved algorithms W_i (with flags) compared to the best algorithm V and the averaged top 10 evolved algorithms V (without flags)

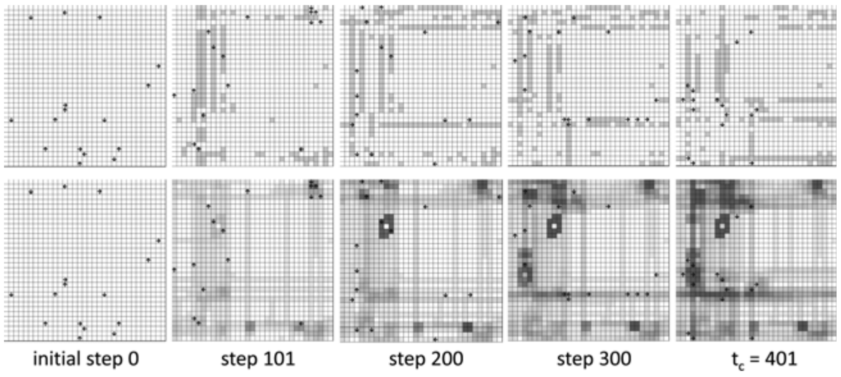


Fig. 6. Patterns of the flags (first line) and the visited cells (second line) for the best algorithm W_4 during the simulation of one environment

the all-to-all communication task, and if the agents are only able to set flags to $f = 1$ when changing their direction, the information provided by the flags cannot be communicated so efficiently.

We have also investigated the patterns of the communication flags. An example is shown in Fig. 6. The number of flags with value $f = 1$ increases during the first steps of the simulation and then it stays at a certain level with some fluctuations in the amount and positions. We observed that the agents very often turn their direction after detecting a flag. Until now it is not clear how to interpret the meaning of the flags in terms of human knowledge.

5 Conclusion

The agents' behavior to solve the all-to-all communication problem by moving around in a cellular automata grid was evolved by a genetic algorithm. We examined the potential of different sets of possible moving and communicating abilities of the agents.

The best evolved algorithm with the basic moving capabilities only needs on average 638 steps to solve the problem. By adding more sophisticated moving abilities and a simple indirect form of communication between the agents we could evolve an algorithm that is able to solve the same problem in only 320 steps on average. In general the algorithms with more sophisticated abilities have a superior behavior and the optimization time is even lower than for the simpler algorithms. Thus we conclude that adding the ability to walk straight and communicate indirectly through flags is an adequate way to improve the agents' behavior.

As the increase of possible actions or states always includes less complex automata, one question that arises is: Is there an amount of states or possible actions at which the efficiency of the genetic algorithm decreases? In future investigations we want to find out whether these limits exist and furthermore if it is a good strategy to evolve at first less complex automata and then use them as a base to develop by hand or by heuristic methods more complex automata with better behavior. We will also focus on the parameters of the genetic algorithm and evaluate other heuristic methods in order to find out which is most suitable for the optimization of state tables. Eventually the set of actions itself could become a candidate for being optimized by heuristic methods.

References

1. 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)
2. Olfati-Saber, R., Fax, J.A., Murray, R.M.: Consensus and cooperation in networked multi-agent systems. Proceedings of the IEEE 95, 215–233 (2007)
3. Lin, J., Morse, A.S., Anderson, B.D.O.: The Multi-Agent Rendezvous Problem. An Extended Summary. In: Nehmer, J. (ed.) Experiences with Distributed Systems. LNCS, vol. 309, pp. 257–289. Springer, Heidelberg (1988)
4. Principe, G., Santoro, N.: Distributed Algorithms for Autonomous Mobile Robots. In: 4th IFIP International Conference on TCS. IFIP, vol. 209, pp. 47–62. 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, June 12-14, pp. 398–410 (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. Sipper, M.: Evolution of Parallel Cellular Machines. LNCS, vol. 1194. Springer, Heidelberg (1997)

8. Sipper, M., Tomassini, M.: Computation in artificially evolved, non-uniform cellular automata. Theor. Comput. Sci. 217(1), 81–98 (1999)
9. 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)
10. Dijkstra, J., Jessurun, J., Timmermans, H.J.P.: A multi-agent cellular automata model of pedestrian movement. In: Schreckenberg, M., Sharma, S.D. (eds.) Pedestrian and Evacuation Dynamics, pp. 173–181. Springer, Heidelberg (2001)
11. Nagel, K., Schreckenberg, M.: A cellular automaton model for freeway traffic. J. de Physique 2, 2221 (1992)
12. Mesot, B., Sanchez, E., Peña, C.A., Perez-Uribe, A.: SOS++: Finding smart behaviors using learning and evolution. In: Standish, R., Bedau, M., Abbass, H. (eds.) Artificial Life VIII: The 8th International Conference on Artificial Life, pp. 264–273. MIT Press, Cambridge (2002)

The GCA-w Massively Parallel Model

Rolf Hoffmann

Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany
hoffmann@ra.informatik.tu-darmstadt.de

Abstract. We introduce the GCA-w model (**G**lobal **C**ellular **A**utomata with **w**rite access) that is an extension of the GCA (Global Cellular Automata) model, which is in turn an extension of the cellular automata (CA) model. All three models are called "massively parallel" because the models are based on cells that are updated synchronously in parallel. In the CA model, the cells have static links to their local neighbors whereas in the GCA model, the links are dynamic to any global neighbor. In both models, the access is "read-only". Thereby no write conflict can occur which reduces the complexity of the model and its implementation. The GCA model can be used for many parallel problems that can be described with a changing global (or locally restricted) neighborhood. The main restriction of the GCA model is the forbidden write access to neighboring cells. Although the write access can be emulated in *O(log n)* time this slowdown is not desired in practical applications. Therefore, the GCA-w model was developed. The GCA-w model allows to change the state of the own cell as well as the states of the neighboring cells. Thereby parallel algorithms can be executed faster and the activity of the cells can be controlled in order, e.g., to reduce power consumption or to use inactive cells for other purposes. The application of the GCA-w model is demonstrated for some parallel algorithms: pointer inversion, sorting with pointers, synchronization and Pascal's triangle. In addition, a hardware architecture is outlined which can execute this model.

Keywords: Massively Parallel Model, Global Cellular Automata, GCA with Write Access, Dynamic Neighborhood, Dynamic Cell Activation, GCA-w Applications, GCA-w Architecture.

1 Introduction

A new massively parallel computing model, called "GCA-w" (GCA with write-access) is proposed [1], that is an extension of the GCA (Global Cellular Automata) model [16, 15, 14, 12, 11] that is in turn an extension of the cellular automata (CA) model. The cells of a GCA can dynamically establish links to their global neighbors, whereas the cells of a CA use fixed links to their local neighbors. The GCA and CA models have in common that they allow only read access to their neighbors and therefore no write conflicts can occur. Thereby the complexity of the model is kept low and implementations in software or parallel hardware can easily be accomplished.

It was already shown that the GCA model is suited for a large number of parallel problems (Jacobi iteration to solve a system of linear equations [9, 3], Finding the connected components of a graph [4, 10], Random Distribution of particles with non local dynamic neighbors [5], N-body force calculation [2], Sorting numbers [8], Graph algorithms [12]). Also efficient parallel hardware architectures [13, 8, 5, 3, 2] have been designed. The language GCAL [9] was developed to simulate GCA algorithms and to use the language as an input for an automatic design process generating an application specific data parallel hardware to be configured on an FPGA.

The GCA model can also be mapped onto the PRAM-CROW model [6]. Therefore, PRAM-CROW algorithms can be executed on the GCA model with the same time complexity using the same amount of processors respectively cells.

The restrictions of the GCA model are twofold:

(1) *No write access to the neighbors*: Although it is possible to simulate a write access in *O(log n)* time [6], this slowdown might be unacceptable for practical applications. In addition, particular algorithms can be described more naturally if the neighbor's state can be modified.

(2) *No dynamic activation*: In the GCA model cells can deactivate themselves. Thereby the number of active cells that are involved in the computation can be reduced. An inactive cell cannot change its state but its state can be read by another cell. Enlarging the number of active cells dynamically is only achievable by an additional mechanism like a central control. In order to control the number of active cells in a decentralized way by the cells themselves, write access to the neighbors is mandatory. The reason is that an inactive cell cannot activate itself; it has to be activated by another active cell. A dynamic varying activity is very often an inherent property of parallel algorithms which should be exploited in order to use the computational resources of inactive cells for other computations or to reduce the power consumption.

Related Work. The PSA model [7] of computation is a very general and powerful model based on *substitution rules*. It allows also modifying the state of arbitrary target cells (*right side* of the substitution) using a "*base*" and a "*context*". In relation to the GCA-w the base corresponds to the cell under consideration, the context corresponds to the read neighbors and the right side corresponds to the cells which are modified. – There is also a relation to the CRCW-PRAM [18, 17] model. The PRAM model is based on a physical view with *p* processors that have global memory access to physical data words whereas the GCA-w is based on logical computing cells tailored to the application. Another difference of the GCA-w model compared to PRAM is the direct support of dynamic links and the rule based approach similar to the CA model.

2 The GCA-w Model

The GCA-w model overcomes the restrictions of the GCA model by allowing write access to the neighbors. A cell can operate in two modes:

(1) *Normal GCA mode (n-mode):* A cell reads information from the dynamically linked neighbors and then updates its own state (data and link information) only.

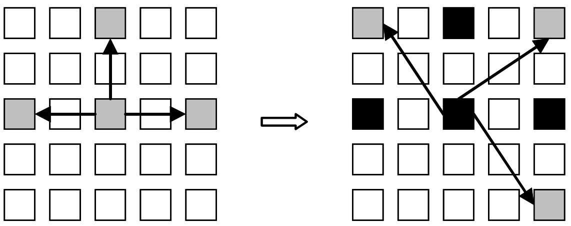


Fig. 1. Each cell is dynamically connected to global (or locally restricted) neighbors (grey). The state of the centre cell including the links and the states of its neighbors can be changed (grey to black) by a local rule.

(2) *Write mode (w-mode):* A cell reads information from the dynamically linked neighbors and then updates its neighbors' states and optionally its own state (Fig. 1).

The w-mode can be used to activate or to inactivate a neighbor, e.g., by sending a certain control code to that neighbor. An inactive cell serves as a storage-only cell that can be accessed (read and write) by another active cell. As an inactive cell does not use its computational resources itself, they might be used by other cells.

With the availability of the w-mode, many parallel algorithms can be described with a lower time-complexity and furthermore the computing resources of inactive cells could be used for other computations.

An inherent problem remains, which is complicating an implementation: There may occur write conflicts. They can either be avoided by using the w-mode in an "exclusive" way meaning that the algorithm ensures that no write conflict can occur. Otherwise, the conflicts have to be resolved automatically in a defined manner. Well-known conflict resolution strategies among others are Arbitrary, Priority, Common, or Reduction. The handling of conflicts will be discussed in more detail in a future contribution.

GCA-w with Unstructured State. A GCA-w consists of an array of processing cells (Fig. 2). Each cell k contains a state q, an address function h and a rule function f. The cells' states are updated in four phases:

(1) The effective address p_{eff} of the global neighbor (in the general case multiple neighbors are permitted) is computed.
(2) The dynamic link to the neighbor is established in order to read state q^*.
(3) The local rule f is applied that computes the results $f1$ and $f2$.
(4) The result $f1$ is optionally written to update the cell's state q and the result $f2$ is optionally written to update the state q^* of the neighbor cell.

Optionally the functions h and f may take into account central control information, like the generation counter t, common parameters, control codes or address offsets. Note that the model does not require central control information because the computation of such information can be replicated in each cell. The reason for using a central control is to minimize the cell's complexity.

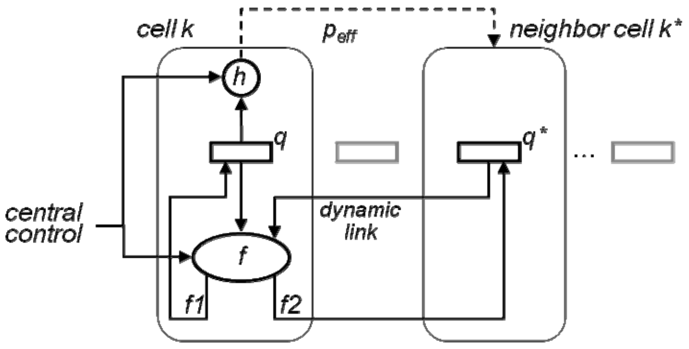


Fig. 2. The global neighbor is accessed using the effective address p_{eff} computed by the cell. The next states *f1* and *f2* are computed and are stored in *q* and *q**.

GCA-w with Structured State. Each cell (Fig. 3) contains a *data field d* and one or more *link information fields p*. The link information field *p* is also denoted as *pointer field* because it can directly act as a pointer if *h* is the identity function. The GCA-w model is called *one-handed* if only one neighbor can be addressed, *two-handed* if two neighbors can be addressed and so on. (In a more general case the number of links could be different for different cells.) The one-handed model seems to be sufficient for most practical applications, as it is the case for most of the GCA algorithms investigated so far. In addition, the multi-handed model can be simulated on the one-handed model. Therefore, the following considerations are restricted to the one-handed model.

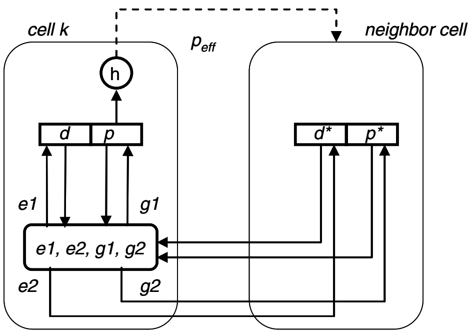


Fig. 3. The state *q* of a cell can be partitioned into a data field *d* and one or more link information fields *p*

The local information *(d, p)* and the global information *(d*, p*)* are inputs of the functions *e* and *g* which compute the next data and the next link respectively (Fig. 3). All cells are updated in parallel in a synchronous way generation by generation. The functions *e* and *g* may further depend on the local space index *k* of the cell and central control information. An optional function *h* is used to compute the effective address p_{eff} of the global cell in the current generation.

The operation principle of a GCA-w can be defined in two forms: *basic* GCA-w model, or *general* GCA-w model that includes the basic model.

Basic Model. The basic GCA-w model removes the address calculation function h (h is the identity function $h = p$), meaning that the effective address is $p_{eff} = p$. The next pointer and the next data fields are computed by the following rules:

$$d <= e1(d, p, d^*, p^*, k, control),$$
$$p <= g1(d, p, d^*, p^*, k, control),$$
$$d^* <= e2(d, p, d^*, p^*, k, control),$$
$$p^* <= g2(d, p, d^*, p^*, k, control).$$

The assignment symbol "<=" is used to indicate the synchronous updating. If the arrays $D = d_0 d_1 \ldots d_{n-1}$ *and* $P = p_0 p_1 \ldots p_{n-1}$ are used to denote the data fields respectively the pointer fields of the cells then d, d^*, p, p^* are equivalent to

$$d = D[k], p = P[k], d^* = D[p] \text{ and } p^* = P[p].$$

Note that the uniform functions used in the cells (not dependent on the local space index k) can be specialized resulting in non-uniform functions: $f(k, \ldots) \rightarrow f_k(\ldots)$.

The advantage of the basic model is that it is simpler because it does not require the function h, thereby also reducing the implementation effort. The user of this model has to be aware that the effective address in the current generation has to be computed in the previous generation. This can always be accomplished if the actual central control information is not influencing directly the effective address. The general GCA model is a more convenient choice from the programmer's point of view when an algorithm can be described more naturally by computing the effective address in the current generation.

General Model. The general GCA-w model uses the additional rule h that computes the effective address:

$$p_{eff} = h(d, p, k, control), d^* = D[p_{eff}], p^* = P[p_{eff}].$$

The assignment symbol "=" indicates that a value is assigned to the temporary pointer variable p_{eff} that may be a wire or a temporary register in a hardware implementation.

Note that a cell of the basic GCA-w can be seen as a Moore-automaton, because the effective address (the output) is stored in a register and does not directly depend on an input. In contrast, a cell of the general GCA-w operates as a Mealy-automaton if the computation of the effective address is depending on central control information received from the environment.

A hardware implementation of the GCA-w model can be simplified in all cases where the pointer p follows an address pattern which is known in advance and which is not data dependent: $p_{eff} = h(k, control)$. Such a case with "known pointers" appears in many applications (e.g., hypercube algorithms). Then the link fields in the cells are not necessary. In a sequential implementation of the model a central address generator can easily generate these addresses.

A structured GCA-w can be transformed into an unstructured GCA-w by unifying the two fields *d* and *p* into one single word *q*, such that *q* is partly or alternatively interpreted as data or pointer information. The unstructured unified model has the advantage that it is simpler and can be implemented with fewer hardware resources. The unstructured model can also been seen as “untyped” because the types “data” and “pointer” are not distinguished whereas the structured model can be seen as “typed”.

Write Conflicts. As mentioned before write conflicts may occur, for different reasons. A cell may use itself as a neighbor if the effective address is equal to the cell's index. If this case is not forbidden or given another semantic then two the values *f1* and *f2* (both produced by the cell itself) to be written may cause a conflict. The probably most frequent conflict might occur if one or more cells *k* try to update a neighboring cell *k** which also tries to update itself. For the one-handed GCA-w at most *n* write conflicts may occur on one cell. If the absence of write conflicts cannot be guaranteed, which is especially the case if the neighborhood is data dependent or random, the model and the corresponding hardware have to offer a defined conflict resolution strategy. We will not discuss here the different possibilities but it is obvious that the conflict detection and resolution will complicate the hardware and slow down the computation.

Modifications of the GCA-w Model or Future Extensions of the Model. The model can be modified or supplemented by further features in order to meet practical, dedicated or more general requirements. Such modifications or features are

- The cell array contains storage-only cells that cannot compute and thus cannot be activated. They are distinguished into "constant" cells (with read-only access) and "variable" cells (with read and write access).
- Cells are dynamically restricted to read access only.
- Cells may be deactivated forever.
- Another updating scheme is used such as asynchronous updating.
- Several cell arrays (interacting or not interacting) are computed in parallel.
- The output of one generation is written into a new cell field in a dataflow manner.
- Non-uniform cells are used built from structures or rules that are space-dependent.
- The cell’s state is separated into a public and private (hidden) part. Only the public part can be accessed by another cell.
- Cells are considered as objects containing methods which can be invoked by the cell itself or by another cell.
- Cells are dynamically created or deleted.

3 Some Applications

It was already shown that the GCA model is applicable to many parallel applications and that it can efficiently be supported by hardware [13, 11]. Recently it was shown that the classical PRAM models can be simulated on the GCA [6]. The write access available in the PRAM models can be simulated with a slowdown of *O(log n)* using a tree of cells. If an algorithm guarantees the “exclusive-write” property this slowdown can be eliminated through the direct write access available in the GCA-w model. In the following, the expressiveness of the GCA-w model is demonstrated by some applications.

Pointer Inversion. Consider an unidirectional linked list. The task is to invert the direction of the pointers. This task can directly be performed through the GCA-write mode.

Version 1: It is assumed that the list is cyclic. Then the algorithm is very simple.

```
type cell = (data: integer; pointer: 0..n-1)
C: array [0 .. n-1] of cell
parallel C[k=0 .. n-1]
   // pointer is directly used as effective address "neighbor"; basic model
   neighbor = pointer
   // neighbor.pointer is an abbreviation for C[C[k].pointer].pointer
   neighbor.pointer <= k  // write own index to neighbor's pointer
endparallel
```

This algorithm can also be used for a non-cyclic linear list. The idea is that one node (e.g., node 0) of the cyclic list is a special node used to mark the head and the tail of the list. An outgoing arc from that node points to the head, an incoming arc marks the tail. The following version is one algorithm for a non-cyclic linear list. Other alternative algorithms are given in [1].

Version 2: This version solves the problem in one generation without additional variables using a write-conflict resolution method based on priorities. The tail of the list is marked by a self-loop. A write conflict occurs for all cells except for the new head cell. This conflict is resolved by giving priority to the write-access induced by a remote cell.

```
type cell = (data: integer; p: 0 .. n-1)
C: array [0 .. n] of cell
initial C[k = 0 .. n-2].p <= k+1, C[n-1].p <= n-1 endinitial
parallel C[k = 0 .. n-1]
      neighbor = p
      conflict  // this section describes the conflict resolution
      // high priority: write k to neighbor's pointer
      priority 1: if (neighbor ≠ k) then neighbor.p <= k endif
      // low priority: mark new head
      priority 2: p <= k  // applies only for new head
      endconflict
endparallel
```

Sorting with Pointers. Some PRAM sorting algorithms and the GCA algorithm in [14] compute for each element first the target position where it should be located. In the second phase, the elements are copied in parallel to their targets. The second phase requires write access to an arbitrary cell. The GCA-w model provides this capability. Note that the following sorting algorithm with *O(n)* steps is not optimal.

```
type cell = (d: integer; target: 0 .. n-1)
parallel C [k = 0 .. n-1] target <= 0 endparallel
// phase 1
for t = 1 to n – 1 do
    parallel C[k = 0 .. n-1]
        neighbor = (k + t) mod n  // defines function h, general model
        if (neighbor.d < d)  or (neighbor.d = d and neighbor < k)
        then target <= target +1 endif
    endparallel
endfor
// phase 2
parallel C[k = 0 .. n-1]  target.d <= d  endparallel  // write to target cell
```

Synchronization. All cells shall change simultaneously from the ZERO state into the FIRE state. The problem is related to the well-known Firing-Squad problem. The number *n* of cells is not known in advance. The "general" is located on the left (*k=0*, Fig. 4) end and starts the process being the only active cell. Using the global neighborhood from the beginning the problem could be solved trivially if all the soldiers (the other cells) would directly observe the general. But it is assumed that at the beginning only local neighborhoods are allowed. Initially each cell is connected to its right neighbor except the right end soldier who is pointing to himself thereby marking the end of the chain (Fig. 4 A). The purpose of the first version with $N+1$ generations is to show how $N = n+1$ cells can be activated in principle one after the other forming a wave of activity. Note that a more efficient algorithm with $2+log_2 N$ generations can be designed using the well-known pointer-jumping technique (version 2, Fig. 4 B).

Version 1

```
type cell = (data: (ZERO, FIRE); p: 0 .. n;  active: activity)
C: array [0 .. n] of cell
initial
    C[0 .. n-1].data <= ZERO
    C[0].active <=  TRUE          // only left border cell is initially active
    C[k = 0 .. n-1].p <= k+1      // cells point to its right neighbor
    C[n].p <=n                    // right border cell points to itself
endinitial

repeat
    parallel C[k where active]       // only do for active cells
         // activate right neighbor, write mode
        if (p = k+1) then p.active <= TRUE endif
        // if not right border reached increment pointer for active cells only
        if (p.p  ≠   p) then p <= p+1 endif
         // wait one step until right border cell was activated
        if (p.p = p) and (p.active = TRUE)  then data <= FIRE endif
    endparallel
endrepeat
```

Version 2

```
type cell = (data: (ZERO, FIRE); p: 0 .. n;  active: activity)
C: array [0 .. n] of cell
initial
     C[0 .. n-1].data <= ZERO
     C[0 .. n-1].active <= TRUE      // except cell n: all are active
     C[k = 0 .. n-1].p <= k+1        // cells point to its right neighbor
     C[n].p <=n                      // right border cell n points to itself
endinitial

repeat
     parallel C[k where active]      // only do for active cells
          p <= p.p                   // pointer jumping
          // if general points to the last soldier n then activate him
          if (k = 0) and (p.p = p) then p.active <= TRUE endif
          // when last soldier was activated then fire all
          if (p.p = p) and (p.active = TRUE)  then data <= FIRE endif
     endparallel
endrepeat
```

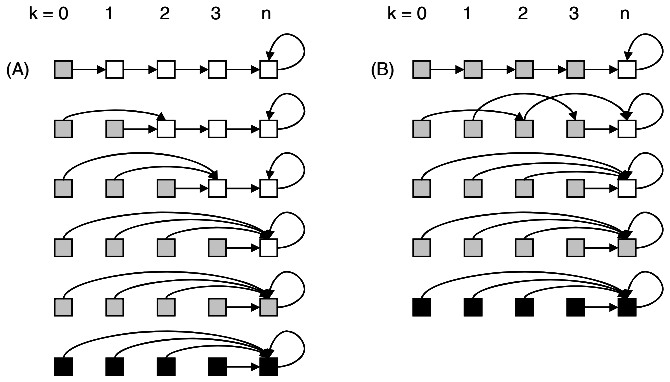


Fig. 4. **(A)** The activity is gradually propagated (*grey*) until after *n+1* generations all cells change into the FIRE state (*black*). **(B)** The last soldier is detected faster using pointer jumping.

Pascal's Triangle. This well-known triangle (already known before Pascal described it, e.g. by Yanghui), computes the binomial coefficients. It can be drawn in different ways. Filling empty sites with zeroes (Fig. 5 A), the computation can easily be formulated by the Cellular Automata rule: (*Center* $\leftarrow$ *Left* + *Right*). However, redundant "zero" additions are performed. In order to avoid unnecessary zero additions, the aligned representation (Fig. 5 B) can be used. Filling the empty sites with zeroes the computation can be described by the CA rule (*Center* $\leftarrow$ *Left* + *Center*) more efficiently.

Note that in a CA with *n* cells always (*n-1*) cells are updated, although in this example the interesting information is propagated gradually from left to right. Only in the last generation, all cells produce a useful result (except cell 0). On average,

(A)

0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	1	0	0	0	0	0
0	0	0	0	1	0	2	0	1	0	0	0	0
0	0	0	1	0	3	0	3	0	1	0	0	0
0	0	1	0	4	0	6	0	4	0	1	0	0
0	1	0	5	0	10	0	10	0	5	0	1	0

(B)

k=0	1	2	3	4	5
1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

Fig. 5. **(A)** The value "0" is used to indicate the spaces. The value x[k] in line j is the sum of x[k-1]+x[k+1] of line (j-1). (1D-CA rule: Center ← Left + Right). **(B)** Aligned representation, result of deleting the "0"s from Fig. 5 A and shifting the remaining numbers to the left border.

approximately only half of the cells are producing a useful result. Of course, by the use of a central control, the cells can gradually be included into the computational process. But it is desired to control the amount of active cells by the cells themselves in a decentralized way. The GCA-w model supplies this feature. The redundant production of zeroes in the right part of a line can be avoided. In the following program only useful computations are performed.

Initially only cell (*k=0*) is active. In the first generation, it performs three actions: (1) writing *d <=1* to its right neighbor (*k=1*), (2) activating the right neighbor, and (3) deactivating itself. Thereby in every new generation, a new cell is activated at the right end.

```
type cell = (data: integer; active: activity)
// left, right are temps used as effective addresses to be computed
C: array [0 .. n] of cell            // example n=5
initial
    C[0].data <= 1, C[0].active <= TRUE
endinitial
repeat
    parallel C[k where active]     // only do for active cells
     left = k-1, right = k+1        // fixed local neighbors like in CA
     if (k=0) then active <= FALSE, right.data<= 1, right.active <= TRUE endif
     if (data ≠ 1) then data <= data + left.data endif
     if (data =1) and (k<n)  then data <= data + left.data,
                                    right.data <= 1, right.active <=TRUE endif
     if (data =1) and (k=n)  then data <= data + left.data,   right.data <= 1 endif
   endparallel
endrepeat
```

4 Hardware Architecture

Efficient hardware architectures can be designed for the GCA-w model with moderate effort. The designer may choose a fully parallel architecture, or a multiprocessor architecture, or a pipelined sequential architecture or a data parallel architecture composed of multiple sequential architectures.

The fully parallel architecture for *n* cells uses *n* registers (*q*, or *d* and *p*), *n* functional units (*h, e, g*) and a link network that can supply in parallel the links which are demanded by the execution of the parallel algorithm. In most of the practical applications the full connectivity (all-to-all) needs not to be implemented because parallel algorithms mainly use only dedicated communication patterns. A standard multiprocessor architecture can be used or designed that is complemented with special hardware components or special vector-like instructions in order to support the model and accelerate the execution. In the following the principle of a sequential pipelined architecture is described that can be used as a component for a data parallel architecture.

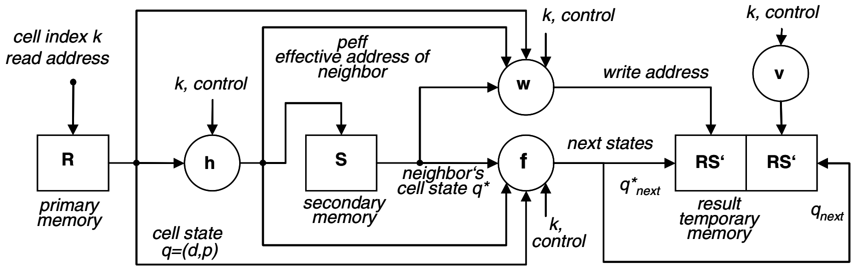


Fig. 6. A general architecture for the GCA-w model. The cells are processed sequentially in a pipeline mode.

The proposed pipelined sequential architecture (Fig. 6) works as follows: The cell's state *q* is read sequentially from the primary memory *R* using an address counter *k*. Then follows the computation of the effective address p_{eff} and the neighbor is read from the memory *S*. Now all the information is available to compute the next cell states q_{next} and q^*_{next} by the function *f* and the write address by the function *w*. *RS'* is a temporary two-port memory which stores q_{next} at the cell's location *k* (or more general at *v*(*k, control*)) and q^*_{next} at the location of the computed write address *w*. *RS'* is copied back into *R* and *S* at the end of the current generation in order to achieve the synchronous updating scheme. In order to avoid the copying of the new data two two-port memories *RS (read+read)* and *RS' (write+write)* can be used which are interchanged by switches after each generation. Alternatively one 4-port memory (2*read + 2*write) can be used that is divided into two pages which are interchanged using the most significant address bit.

Note that the write address will be equal to the neighbor's address ($w = p_{eff}$) for most applications but the hardware offers a more general functionality $w = w\ (q, p_{eff}, q^*, k, control)$, e.g., $w = k+constant$, $p+constant$, $p_{eff}+constant$, p^*. This architecture can be specialized in several ways, e.g., by using a multiplexer with the inputs *k, k+1, k-1, p,* p_{eff}, p^* for the function *w*. Thereby another "write"-neighbor than p_{eff} can be chosen. If $w=p^*$ the new state q^*_{next} is written to the address stored in the neighbor p_{eff}. This case offers an additional indirect accessing, which might be useful in sophisticated parallel algorithms, e.g., move data to a target that is specified in the neighbor. In addition the write address *v* of the own cell can be modified, typically by using an offset $v = k+offset$. Thereby the cells in the array can be shifted or permuted, or the whole cell field can be placed at another base address in order to realize a dataflow / streaming mode of operation.

A conflict resolution logic has to be added to the architecture in case that a GCA-*w* algorithm running on this architecture induces write conflicts. This logic becomes more complex when the degree of parallelism implemented in the hardware is increasing.

If only active cells shall be computed then additional logic has to be added, e.g., an activation list holding the indices k_{active} of the active cells. The list can be updated in parallel to the cell processing and it has to be used instead of the cell counter *k*.

Compared to a sequential GCA-architecture [13, 5] the write address function *w,* the second write port to the *RS'* memory, and the conflict resolution logic has to be added.

5 Conclusion

A new parallel computing model called GCA-w, Global Cellular Automata with write access to the neighbors, was presented. GCA-w is an extension of the GCA model that is related to the Cellular Automata (CA) model. In the GCA and GCA-w model the neighbors are linked dynamically to the cell under consideration and the data and the link information are modified by a local rule. Thereby a cell can decide itself which shall be its neighbors in the next generation. The new GCA-w model overcomes the restriction that a cell can only modify its own state. Thereby any global cell in the whole cell array can be the target of an information transfer. Compared to the normal GCA model a write access needs not to be simulated with a slowdown of *O(log n),* it can be performed directly in *O(1).* Furthermore "sleeping" cells can be turned dynamically into active cells in a decentralized way. "Sleeping" resources might be assigned dynamically to other active cells leading to better resource utilization or lower power consumption. Classical PRAM algorithms as well as practical parallel applications can easily be mapped onto this model. Such algorithms can also be described clearly with constructs available from classical parallel languages. The proposed architecture shows that hardware support of this model can be realized without much effort.

Acknowledgment. Many thanks to Prof. Keller (FernUni Hagen) who made a number of valuable comments on the manuscript, especially concerning the handling of conflict situations and the pointer inversion example. I also like to thank P. Ediger who carefully read the manuscript and contributed to different versions of the pointer inversion example.

References

1. Hoffmann, R.: GCA-w: Globaler Zellularer Automat mit Schreibzugriff, Fachgebiet Rechnerarchitektur, Technische Universität Darmstadt, Internal Report (January 2009)
2. Jendrsczok, J., Hoffmann, R., Lenck, T.: Generated Horizontal and Vertical Data parallel GCA Machines for the N-Body Force Calculation. In: Berekovic, M., Müller-Schloer, C., Stephan Wang, C.H. (eds.) ARCS 2009. LNCS, vol. 5455, pp. 96–107. Springer, Heidelberg (2009)

3. Jendrsczok, J., Homann, R., Ediger, P.: A Generated Data Parallel GCA Machine for the Jacobi Method, 3. In: HiPEAC Workshop on Reconfigurable Computing, HiPEAC Conf. Cyprus 2009 (2009)
4. Jendrsczok, J., Hoffmann, R., Keller, J.: Implementing Hirschberg's PRAM-Algorithm for Connected Components on a Global Cellular Automaton. International Journal of Foundations of Computer Science (IJFCS) 19(6) (2008)
5. Jendrsczok, J., Ediger, P., Hoffmann, R.: A scalable configurable architecture for the massively parallel GCA model. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), Workshop on Advances in Parallel and Distributed Computational Models (APDCM) (April 2008)
6. Osterloh, A., Keller, J.: Das GCA-Modell im Vergleich zum PRAM-Modell. Informatik-Bericht 350 - 3/2009, FernUniversität in Hagen
7. Achasova, S., Bandman, O., Markova, V., Piskunov, S.: Parallel Substitution Algorithms, Theory and Applications. World Scientific, Singapore (1994)
8. Heenes, W.: Entwurf und Realisierung von massivparallelen Architekturen für Globale Zellulare Automaten. PhD thesis, Technische Universität Darmstadt (2007)
9. Jendrsczok, J., Ediger, P., Hoffmann, R.: The Global Cellular Automata Experimental Language GCA-L, Technischer Bericht, RA-1-2007, Technische Universität Darmstadt, FB Informatik (2007)
10. Jendrsczok, J., Hoffmann, R., Keller, J.: Hirschberg's Algorithm on a GCA and its Parallel Hardware Implementation. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 815–824. Springer, Heidelberg (2007)
11. Heenes, W., Hoffmann, R., Jendrsczok, J.: A Multiprocessor Architecture for the Massively Parallel Model GCA. In: IPDPS/SMTPS 2006, IEEE Proceedings: 20th International Parallel & Distributed Processing Symposium (2006)
12. Ehrt, Chr.: Globaler Zellularautomat: Parallele Algorithmen. Diplomarbeit, Technische Universität Darmstadt (2005)
13. Hoffmann, R., Heenes, W., Halbach, M.: Implementation of the Massively Parallel Model GCA. In: PARELEC, pp. 135–139. IEEE Computer Society, Los Alamitos (2004)
14. Hoffmann, R., Völkmann, K.-P., Heenes, W.: GCA: A massively parallel Model. In: IPDPS 2003 (2003)
15. Hoffmann, R., Völkmann, K.-P., Waldschmidt, S., Heenes, W.: GCA: Global Cellular Automata, A Flexible Parallel Model. In: Malyshkin, V.E. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 66–73. Springer, Heidelberg (2001)
16. Hoffmann, R., Völkmann, K.-P., Waldschmidt, S.: Global Cellular Automata GCA: An Universal Extension of the CA Model. In: Worsch, T. (ed.) ACRI 2000 Conference (2000)
17. Keller, J., Keßler, Chr., Träff, J.: Practical PRAM Programming. Wiley, Chichester (2001)
18. JaJa, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)

Implementation of Fine-Grained Algorithms on Graphical Processing Unit

Konstantin Kalgin

ICMMG SB RAS, Novosibirsk, Russia
kalgin@ssd.sscc.ru

Abstract. In this paper we solve the problem of mapping of fine-grained algorithm to graphical processing unit (GPU). Synchronous, asynchronous, block-synchronous and probabilistic cellular automata and explicit scheme of PDE are used as examples. Different implementation variants and their performances are presented.

1 Introduction

As history has shown, there are two ways of processor development. The first one is to design completely new processor architecture. The second one — to improve and adapt existing processors to new market requirements. Both ways have highs and lows, but it is not the subject of our work. We just note that in the second way the end product is cheaper and comes to the market faster. This point has influenced on the appearance of our paper.

Nowadays processor architecture becomes significantly more complex — multicore processors come instead of conventional ones. Nobody now complicates structures of cores. It is more efficient to make them simpler and then increase the number of cores on a chip. But in this case a processor becomes more complicated.

When a new processor architecture appears, we meet the following problem: how to map existing algorithms into it.

In this paper we solve the problem of mapping of fine-grained algorithm to the GPU. We consider only GPU with CUDA [1] support. Synchronous, asynchronous, block-synchronous and probabilistic cellular automata and explicit scheme of PDE (ES) are used as examples.

Of course, our choice of GPU does not depend on low cost and high availability. We choose it as far as GPU architecture and fine-grained algorithms structure are conformed.

2 GPU Architecture

General points. GPU is a well known device connected to the main board (host computer) through the PCI Express bus. It consists of a processor and an on-board RAM (further called *global memory*). A host program fully controls the

GPU (memory management and program execution). The processor consists of multiprocessors (MP). MP is a union of eight streaming processors (SP). Processor and memory frequencies, number of MPs and global memory size depend on GPU version and may be adopted to market requirements.

Multiprocessor architecture is mainly SIMD-oriented (Single Instruction Multiple Data). But it is more flexible than the majority of conventional SIMD-extensions. Sometimes it is called SIMT (Single Instruction Multiple Threads). SIMT architecture requires a programmer to manipulate the threads rather than the vectors. With not too high restrictions on performance it also permits arbitrary branching behavior for threads.

Memory hierarchy. In addition to on-board global memory there are other types of on-chip memory: registers (8192 or 16384 32-bit registers per multiprocessor), shared memory (16384 bytes per multiprocessor), constant and texture caches. The registers have 0-clock latency, the shared memory has 4 clocks of latency and the global memory has 300-600 clocks of latency. The shared memory is used as a fast temporary buffer for processed data. It helps to decrease the intensity of global memory usage.

Kernel is a C function that, when called, is executed N times in parallel by N different CUDA threads. Every thread that executes a kernel is given a unique *thread ID* (further called *coordinates*) accessible within the kernel.

Thread hierarchy. The whole set of CUDA threads is divided into blocks. Blocks have the same sizes (less then 512) and shapes. Thus, a kernel is executed by equally-shaped thread blocks. So, the total number of threads equals the number of threads per block times the number of blocks. Threads within a block can cooperate by sharing data through some shared memory and synchronizing their execution to coordinate memory access.

3 Cellular Automata and Explicit Scheme

Model spaces of CA and ES are both discrete and regular. Further we consider only 2-dimensional model spaces (cellular arrays and meshes). Values of a function corresponding to ES are said to be located in mesh points, whereas CA states are stored in mesh cells. However, for considered model spaces it does not matter.

The following properties are important for the performance of parallel implementation of the models:

– local interactions — to compute new states/values we need only neighboring cell states/values;
– spatial (data) parallelism — computations of new values can be performed simultaneously.

We give no general definition of CA. It can be found in [4], where not only classical but asynchronous, probabilistic and block-synchronous CA are described. Though, in the examples each case is properly defined.

3.1 Explicit Scheme

Let us consider a differential equation with partial derivatives that models phase separation process [6]:

$$u_t = 0.2(u_{xx} + u_{yy}) - 0.2(u - 0.1)(u - 0.5)(u - 0.9) \tag{1}$$

Equation (1) is solved by ES with five-point template (2):

$$\begin{aligned} u^{t+1}_{x,y} = u^t + \frac{\tau}{5}\Big(&\frac{u^t_{x-1} + u^t_{x+1} + u^t_{y-1} + u^t_{y+1} - 4u^t}{h^2} - \\ &-(u^t - 0.1)(u^t - 0.5)(u^t - 0.9)\Big) \end{aligned} \tag{2}$$

It is interesting that equation (1) has CA analog considered in Sec. 3.2.

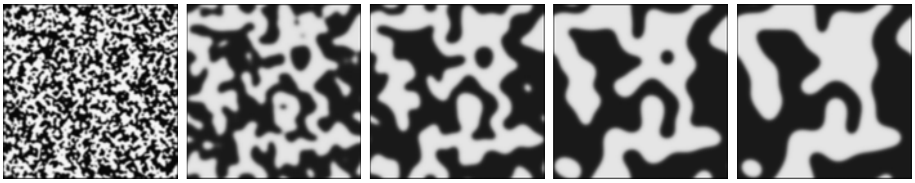


Fig. 1. First, 50th, 100th, 200th and 300th time step of phase separation process simulated by ES (2). Mesh size is 300×300.

Implementation. Every ES time step a grid of threads is spawned having the same size as the ES mesh. The set of mesh point coordinates and the set of thread coordinates are isomorphic. For each mesh point of the model space we put into one-to-one correspondence a GPU thread having identical coordinates.

A mesh point is called *inner* with respect to a block of threads, if the thread corresponding to the point belongs to the block. Remaining mesh points are called *outer* points. A mesh point is called *boundary* point with respect to the block of threads, if it is outer and its value is needed to compute a new value in an inner point.

Implementation with only global memory usage. Values of the function $u^t(x, y)$ in mesh points (further called "values in points") are stored in the global memory. Therefore each use of a value in a point yields global memory access. In addition, a new value computing requires four global memory access. Totally we have *five* global memory access.

Implementation with shared memory usage. Values in mesh points are also stored in the global memory. The shared memory is used only when the block of threads is running. It is used for storing the values in inner and boundary points. Each value in an inner point is loaded to the shared memory from the global memory by appropriate thread. Special threads are assigned for loading values in boundary points to the shared memory.

Thus, in comparison to the above case the number of global memory access is reduced, i.e. values in inner points are loaded from the global memory only *once* and from the shared memory only *five* times per iteration.

3.2 Classical Cellular Automata

Let us consider an example of classical CA that models phase separation process. Each cell can be in one of two states — 0 or 1. The state of it depends on the states of eight neighbors of the cell.

Simulating process consists of iterations. On each iteration we simultaneously update all cell states according to the rule:

$$state = \begin{cases} 0, & sum = 5 \text{ or } sum < 4 \\ 1, & \text{else} \end{cases},$$

where *sum* is the number of neighboring cells in state 1.

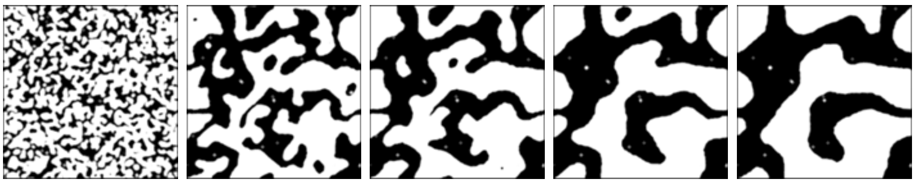


Fig. 2. First, 50^{th}, 100^{th}, 200^{th} and 300^{th} iterations of phase separation process modeling by CA. Mesh size is 300×300.

Implementation of CA differs from ES only by the computations of new values and by the number of used neighboring cells. It is important with a view to GPU implementation performance.

A 32-bit integer is used to represent a cell state in GPU. This choice is caused by GPU constraints on the fast global memory access. With 8-bit or 16-bit integer it results in hardware or software time overhead (packing and unpacking). Experiments show that packing and unpacking lead to a decrease of performance (rather than an increase).

Time of computations significantly depends on neighborhood size because the number of global memory access depends on it. In the case of cellular automata, the number of global memory access to inner cells is reduced by factor 9 (thanks to the shared memory).

3.3 Probabilistic Cellular Automata

Let us consider an example of probabilistic cellular automata (PCA) with Margolus neighborhood that models the process of diffusion. This PCA has the continuous analog — PDE of diffusion (heat conduction).

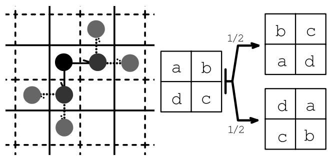


Fig. 3. Possible movements

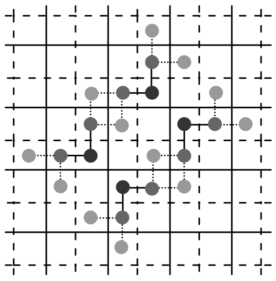


Fig. 4. Possible movements in two iterations

Each cell can be in one of two states — 0 and 1. Cellular array is decomposed into odd and even squares of 2×2 cells (Fig. 3). Simulation process consists of iterations. An iteration consists of simultaneous rotatings of all even squares and then simultaneous rotatings of all odd squares of 2×2 cells (Fig. 4). Each square is rotated independently and clockwise or counterclockwise (Fig. 2). Direction of rotations is determined by pseudo-random numbers.

Implementation of this CA have two special specific properties.

The *first* one is that we need pseudo-random numbers (PRN) to compute new cell states. GPU implementations of PRN generators are considered in Sec. 3.4. Notice that to compute new states of all cells we need to generate the amount of PRNs equal to half a number of cells.

Second. For computing new cell states we need to spawn a grid of threads twice. In this case the amount of calculations is not growing rather than the amount of global memory access. Computation time is growing respectively. We can see similar time increase in the case of block-synchronous CA considered in Sec. 3.6.

3.4 Pseudo-random Number Generation

To generate PRNs with large periods we use two types of generators: 64-bits linear congruential generator (LCG) and 32-bits Mersenne twister (MT). Used particular parameters and means of parallelization are taken from [2,3].

We run N/K GPU threads for generating N numbers: each thread generates K numbers. At the beginning of each thread execution we should load the set of its parameters. This set has a determined size. So, with small K thread initialization takes the most of time. But with large K we have small number of threads, therefore the full power of GPU is not available. The optimal K is selected via experiment.

Main difference between these generators (with a view to implementation performance) is in the number of required operations and in the type of operands. In LCG 64-bit operands are used, whereas in MT 32-bit operands are used. But with MT we have to carry out much more operations then with LCG.

It is shown in Sec. 4.2 that the relation between performances of these generators depends on particular GPU version. So, to gain better performance of number generation with particular GPU the appropriate generator should be chosen.

3.5 Asynchronous Cellular Automata

Let us consider asynchronous cellular automata (ACA) to make full view of capability of fine-grained implementation on GPU.

Classical example is the diffusion ACA. Simulating process of ACA is divided into steps. At each step a cell is chosen randomly. Then its state exchanges with the state of a neighboring cell (again randomly chosen).

This CA has the same continuous counterpart as the considered above PCA — differential equation of diffusion. Since this ACA is probabilistic, computation of new values needs pseudo-random numbers generation.

Implementation. We can not simultaneously execute several steps on the same cellular array without synchronization because these steps may have the same values of cells in use. It can lead to races and wrong results.

Therefore, we have to generate N numbers in special manner to execute N steps without races. This task is not too simple in the sense of computational complexity.

Assume we are able to solve this problem in a negligibly small time. Unfortunately, even under this assumption, GPU can not overtake CPU in simulating the ACA. Results of execution are represented in Sec. 4.3.

3.6 Block-Synchronous Cellular Automata

There is no suitable computer that allows to increase ACA simulating performance easily and efficiently. For ACA simulating on a cluster there are at least three complicated parallel algorithms [7,8,9]. As for GPU, the efficient execution is constrained by random memory access time.

Therefore, block-synchronous cellular automata (BSCA) analog is used instead of the original ACA [4] due to the following facts:

– BSCA can be easily constructed from the particular ACA
– BSCA can be efficiently implemented on majority of computers
– BSCA approximates ACA

Let us consider block-synchronous analog of ACA diffusion [5]. Simulating process consists of iterations. Each iteration has five macro-steps. A macro-step consists of synchronously executed steps.

The coordinates of cells to be updated at each step are determined as follows. The cellular array is covered in regular manner by five-cell template (von Neumann neighborhood) without intersections. The remaining four similar coverings are obtained by translation of the first one. Let us put coverings into one-to-one correspondence with macro-steps. Then, for each macro-step we can choose centers of templates of appropriate covering as steps coordinates. It allows us to avoid races.

Implementation of BSCA slightly differs from considered implementation of PCA. The main difference is in the number of spawned grid of threads per iteration (five times). It results in the corresponding time growing, see Sec. 4.3.

4 Results

In this section performances of different implementations are showed on Core2 processor (one core), GeForce 8800 GTX 512Mb and GTX 280 1Gb (further called GPU 8800 and GPU 280). The unit of performance measurement is *million updated cells/second* (Mcell), and *million generated numbers/second* (Mprn).

4.1 Cellular Automata and Explicit Difference Scheme

There are two main properties of any particular CA and ES with a view to implementation performance: the neighborhood size and complexity of new state/value computation. The CA under consideration has 9-cell neighborhood and ES has 5-cell one. But computation of a new value in ES takes more time then computation of a new state in CA. So, the times of execution for CA and ES are approximately the same.

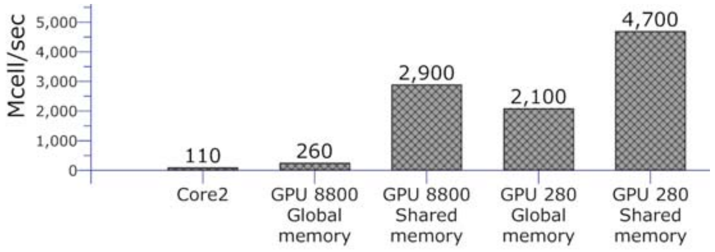


Fig. 5. Cellular automata (nine-point template, 8 operations)

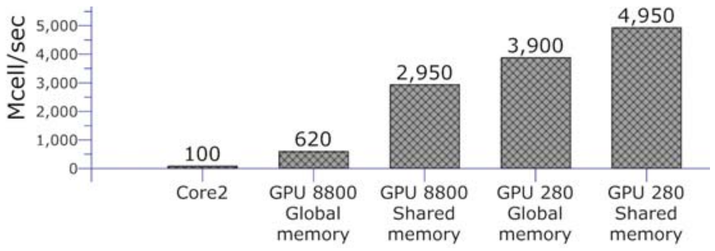


Fig. 6. Explicit scheme (five-point template, 14 operations)

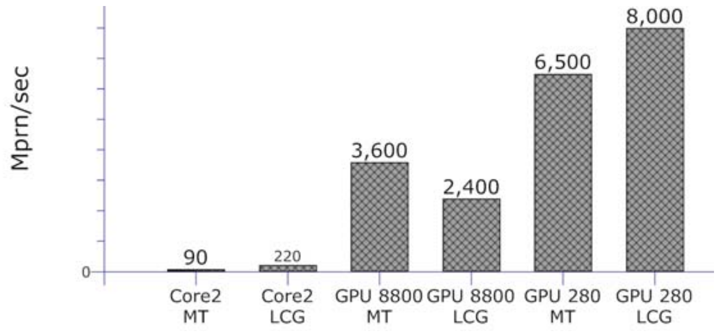


Fig. 7. Pseudo-random numbers generation

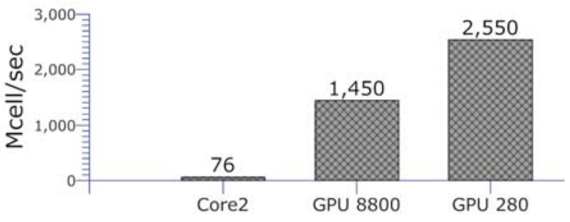


Fig. 8. Margolus diffusion

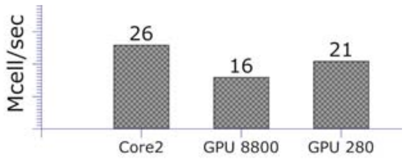


Fig. 9. Asynchronous cellular automata diffusion

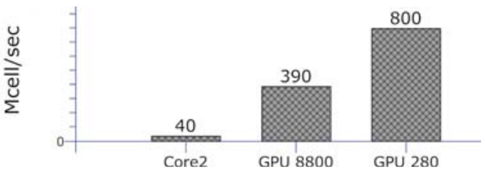


Fig. 10. Block-synchronous cellular automata diffusion

On Figs. 5 and 6 we can see the influence of shared memory usage. Take notice of the difference between performances on GPUs: the influence of shared memory usage is more significant on GPU 8800 than on GPU 280. It can be explained by substantial improvement of global memory access subsystem in GPU 280.

4.2 Probabilistic Cellular Automata

We have implemented two pseudo-random generators: Mersenne Twister (MT) and Linear Congruential Generator (LCG). It is shown in Fig. 8 that ratio of generators performances depends on GPU generation. So, to gain a better performance of number generation with a particular GPU the appropriate generator should be chosen. In Fig. 7 we can see that the PCA execution time equals nearly half of the synchronous CA execution time. Origins of the execution time increasing are described in Sec. 3.3.

4.3 Asynchronous and Block-Synchronous Cellular Automata

The efficient execution of ACA on GPU is constrained by time of random memory access. The time of random memory access is ten times as much as time of regular memory access. These constraints result in poor performance independently of GPU version (Fig. 9). The BSCA implementation shows better results (Fig. 10) because of regular memory access.

5 Conclusion

In this paper we study some aspects of the problem of fine-grained algorithm mapping to GPU. Explicit scheme and different sorts of cellular automata are used as examples. All considered examples, with the exception of ACA, show sufficiently large GPU performances relatively to the CPU performances. Experiments show suitability of GPU usage in explicit-scheme and cellular automata simulating.

References

1. NVIDIA CUDA Programming Guide, `http://www.nvidia.com/object/cuda.html`
2. Podlozhnyuk V.: Parallel Mersenne Twister, `http://www.nvidia.com/object/cuda.html`
3. Marchenko, M.: Parallel Pseudorandom Number Generator for Large-Scale Monte Carlo Simulations. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 276–282. Springer, Heidelberg (2007)
4. Bandman, O.L.: Synchronous versus asynchronous cellular automata for simulating nano-systems kinetics. Bulletin of the Novosibirsk Computer Center. Series: Computer Science (27), 1–12 (2006)
5. Bandman, O.L.: Parallel Simulation of Asynchronous Cellular Automata Evolution. In: El Yacoubi, S., Chopard, B., Bandini, S. (eds.) ACRI 2006. LNCS, vol. 4173, pp. 41–47. Springer, Heidelberg (2006)
6. Schlogl, F.: Chemical reaction models for non-equilibrium phase transitions. Zh. Physik. 253, 147–161 (1972)
7. Lubachevsky, B.D.: Efficient Parallel Simulations of Asynchronous Cellular Arrays. Complex Systems 1(6), 1099–1123 (1987)
8. Overeinder, B.J., Sloot, P.M.A.: Extensions to Time Warp Parallel Simulation for Spatial Decomposed Applications
9. Kalgin, K.V.: Parallel Simulation of Asynchronous Cellular Automata evolution. Bulletin of the Novosibirsk Computer Center. Series: Computer Science (27), 55–62 (2008)

Parallel Implementation of Lattice Boltzmann Flow Simulation in Fortran-DVM Language

Leonid Kamenshchikov

Institute of Computational Modeling SB RAS,
Krasnoyarsk, Akademgorodok, 660036, Russia
lk@icm.krasn.ru

Abstract. During the last twenty years the lattice Boltzmann method (LBM) has been developed as an alternative approach for modeling of fluid dynamics. A parallel implementation of the LBM for 3D fluid dynamics simulations using the Fortran-DVM language is presented. The LBM is parallelized by using spatial decomposition and implemented on a distributed memory cluster MVS-100K. The test problem has been solved for different number of processors (from 1 to 1024). Pictures of flows are compared visually with the similar pictures published in the literature.

1 Introduction

Development of the Lattice Boltzmann method (LB-method) has begun in 1988 when the article of McNamara and Zanetti [1] has been published. They suggested to replace Boolean variables in already well-known discrete models of gases by continuous distribution functions. Since then it has been published many books on LBM, for example [2,3,4], and the number of papers grows exponentially [5–25].

The lattice Boltzmann model is based on the statistical physics and describes the microscopic behavior of particles in a very simplified manner, but on the macroscopic level it gives correct average values for velocity, density, pressure and for other characteristics of flows.

The main advantages of the LB-method are: (1) initial equations have a simple form, in its there are derivatives only of the first order, nonlinearity is present only in an algebraic source; (2) in view of local character of calculations, the LB-method is easily realised on parallel computers that in much "compensates" an increase of total number of the solved equations; (3) processes of convection-diffusion are presented by means of a small set of the fixed velocity vectors of particles that reduces corresponding calculations to a simple shift along these vectors; (4) it is not necessary to solve Poisson equation on each time step in all domain for the correction of pressure as it becomes in many traditional algorithms for solution of the Navier-Stokes equations; (5) many scientists predict to this method the big prospects for modeling of physical and chemical processes in geometrically complex areas of micro and nano sizes (porous media and micro-structures); (6) easiness of a writing of machine programs.

2 Lattice Boltzmann Model

Following [5,6,7] we will consider the basic equations and the simplest numerical algorithm for modeling of isothermal incompressible fluid flows as the most typical case.

The LB-method is based on the classical Boltzmann kinetic equation:

$$\frac{\partial f}{\partial t} + \boldsymbol{\xi} \cdot \nabla_{\boldsymbol{r}} f + \boldsymbol{F} \cdot \nabla_{\boldsymbol{\xi}} f = Q(f), \tag{1}$$

where $f \equiv f(\boldsymbol{r}, \boldsymbol{\xi}, t)$ is a single-particle distribution function in the phase space $(\boldsymbol{r}, \boldsymbol{\xi})$, $\boldsymbol{r} = (x, y, z)$ is the coordinate of a particle, $\boldsymbol{\xi}$ is velocity, t is time, $\boldsymbol{F}$ is external force, $Q(f)$ is integral of collisions.

The macroscopic variables (density, velocities, pressure) can be computed simply by moment integration as

$$\rho(\boldsymbol{r}, t) = \int f d^3\xi, \quad \boldsymbol{u}(\boldsymbol{r}, t) = \frac{1}{\rho} \int \boldsymbol{\xi} f d^3\xi, \quad p(\boldsymbol{r}, t) = \frac{1}{3} \int (\boldsymbol{\xi} - \boldsymbol{u})^2 f d^3\xi. \tag{2}$$

The nonlinear integro-differential equation (1) is too complex for numerical solution. Therefore a success of the lattice Boltzmann method became possible only after invention of the simplified forms of this equation.

First, the nonlinear integral of collisions $Q(f)$ has been written (for the simplest case) in the form of Bhatnagar-Gross-Krook [26]:

$$Q(f) = \frac{1}{\tau}(f^{eq} - f), \tag{3}$$

where

$$f^{eq} = \frac{\rho}{(2\pi RT)^{3/2}} \exp\left(-\frac{(\boldsymbol{u} - \boldsymbol{\xi})^2}{2RT}\right) \tag{4}$$

is the Maxwell-Boltzmann distribution function, τ is the relaxation time due to collision, R is the ideal gas constant, T is temperature.

Second, a finite set (a lattice) of velocities in the space of particles velocities is fixed:

$$\mathcal{L} = \{\boldsymbol{\xi}_m,\ m = 0, \ldots, M-1\}. \tag{5}$$

An example of such lattice for 3D ($M = 19$) space is shown in Fig. 1. The zero vector always enters into the set $\mathcal{L}$.

Denoting $f_m(\boldsymbol{r}, t) = f(\boldsymbol{r}, \boldsymbol{\xi}_m, t)$ and suppose for simplicity $\boldsymbol{F} = \boldsymbol{0}$, we receive the system of discrete Boltzmann equations, which are required to be solved numerically:

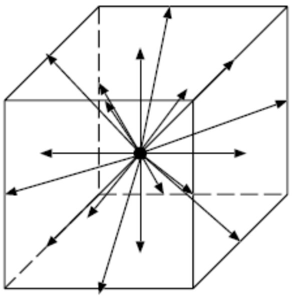


Fig. 1. Cubic velocity lattice $\mathcal{L}$ for 3D models ($M = 19$)

$$\partial_t f_m + \boldsymbol{\xi}_m \cdot \nabla_{\boldsymbol{r}} f_m = \frac{1}{\tau}(f_m^{eq} - f_m), \quad m = 0, \ldots, M-1. \tag{6}$$

It is also supposed that a uniform grid, with an identical step δ_x on all axes, in the space of coordinates is given:

$$\mathcal{P} = \{\boldsymbol{r}_{ijk} = \boldsymbol{r}_0 + [(i-1)\delta_x, (j-1)\delta_x, (k-1)\delta_x], \\ i = 1, \ldots, n_i, \quad j = 1, \ldots, n_j, \quad k = 1, \ldots, n_k\}. \tag{7}$$

Integrating (6) on a small interval $(0, \delta_t)$ along the line $\boldsymbol{\xi}_m$ and neglecting by the variables of the order $O(\delta_t^2)$ we receive the lattice Boltzmann equation (LB-equation):

$$f_m(\boldsymbol{r} + \boldsymbol{\xi}_m \delta_t, t + \delta_t) = f_m(\boldsymbol{r}, t) + \frac{\delta_t}{\tau}(f_m^{eq} - f_m), \quad m = 0, \ldots, M-1, \tag{8}$$

or denoting $\omega = \delta_t/\tau$:

$$f_m(\boldsymbol{r} + \boldsymbol{\xi}_m \delta_t, t + \delta_t) = \omega f_m^{eq}(\boldsymbol{r}, t) + (1-\omega) f_m(\boldsymbol{r}, t), \quad m = 0, \ldots, M-1. \tag{9}$$

The equations (9) are usually solved by splitting method in two steps [7]. On the first step the results of collisions only are calculated:

$$\widetilde{f_m}(\boldsymbol{r}, t) = \omega f_m^{eq}(\boldsymbol{r}, t) + (1-\omega) f_m(\boldsymbol{r}, t), \quad m = 0, \ldots, M-1, \quad \boldsymbol{r} \in \mathcal{P}. \tag{10}$$

On the second step the propagation process (or 'free flight' [9]) is considered:

$$f_m(\boldsymbol{r} + \boldsymbol{\xi}_m \delta_t, t + \delta_t) = \widetilde{f_m}(\boldsymbol{r}, t), \quad m = 0, \ldots, M-1. \tag{11}$$

The mass density and velocity vector are defined as follows:

$$\rho(\boldsymbol{r}, t) = \sum_m f_m(\boldsymbol{r}, t) \quad \boldsymbol{u}(\boldsymbol{r}, t) = \frac{1}{\rho} \sum_m \boldsymbol{\xi}_m f_m(\boldsymbol{r}, t). \tag{12}$$

It has been shown [6] that these $\rho(\boldsymbol{r}, t)$ and $\boldsymbol{u}(\boldsymbol{r}, t)$ are also solutions of following the Navier-Stokes equations:

$$\nabla \cdot \boldsymbol{u} = 0, \tag{13}$$

$$\rho \partial_t \boldsymbol{u} + \rho \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\nabla p + \nu \rho \nabla^2 \boldsymbol{u}, \tag{14}$$

where

$$p = c_s^2 \rho, \quad c_s = c/\sqrt{3}, \quad c = \delta_x / \delta_t, \quad \nu = \frac{(\tau^* - 1/2)}{3} \frac{\delta_x^2}{\delta_t}, \quad \tau^* = \tau/\delta_t. \tag{15}$$

At derivation of the equations (13)–(14) all expressions of the order $O(\delta_x^2)$, $O(Ma^2)$ and above have been neglected. Here $Ma = |\boldsymbol{u}|/c_s$ is the Mach number which should be small enough to provide incompressibility of fluid. For this case and for constant temperature the equilibrium distribution functions can be written in more simple form [5]:

$$f_m^{eq} = \rho w_m \left[1 + \frac{\boldsymbol{\xi_m} \cdot \boldsymbol{u}}{c_s^2} + \frac{1}{2c_s^4} \Big((\boldsymbol{\xi_m} \cdot \boldsymbol{u})^2 - c_s^2 |\boldsymbol{u}|^2 \Big) \right], \quad m = 0, \ldots, M-1, \tag{16}$$

where

$$w_0 = 1/3; \quad w_m = 1/9, \ m = 1, \ldots, 6; \quad w_m = 1/36, \ m = 7, \ldots, 18. \tag{17}$$

3 Fortran-DVM Language

Usage of well-known MPI-approach for parallel programming has some significant disadvantages: the development and debugging of such programs requires much more effort from the programmer because the level of language is too low; the efficient program execution on clusters requires load balancing which is difficult to provide in MPI-approach, etc.

DVM-system developed in Keldysh Institute of Applied Mathematics of Russian Academy of Sciences allows to develop parallel programs in C-DVM and Fortran-DVM languages for different architecture computers and computer networks [27,28,29].

Fortran-DVM is a set of extensions to the Fortran-77 standard that permits programmers to distribute data among multiple processors. Using Fortran-DVM languages a programmer deals with the only one version of the program both for serial and parallel execution. Besides algorithm description by means of Fortran-77 features the program contains rules for parallel execution of the algorithm. These rules are syntactically organized in such a manner that they are "invisible" for standard Fortran compilers and doesn't prevent DVM-program execution and debugging on personal computers or workstations as usual serial program.

The main goals of the DVM-system are follows [29]. Simplicity of parallel program development. Portability of parallel program onto different architecture computers (serial and parallel). For serial computers the portability is provided by DVM-directive 'transparency' for standard Fortran-77 compilers. High performance of program execution. Reusability (composition of parallel applications from several modules). Unified parallelism model for Fortran-77 languages, and, as result, unified system of runtime support, debugging, performance analyzing and prediction. Elements of Fortran-95 are allowed also at the latest versions.

A user of DVM-system always receives following characteristics of productivity of the program [29]: (1) the number of used processors N_{cpu}; (2) astronomical execution time; (3) 'total processor time' is production of the time of execution by the number of used processors; (4) 'productive time' is a predicted execution time on one processor; (5) 'efficiency coefficient' is ratio of productive time to total processor time; (6) 'lost time' is total processor time minus productive time. Lost time components: (6a) 'insufficient parallelism' are losses because of performance of serial parts of the program on all processors; (6b) 'communications' are losses because of interprocessor exchanges; (6c) 'idle time' is time of absence any operation on processors; (7) 'load imbalance' is possible losses because of different loading of processors.

Calculations were performed on cluster MVS-100K at the Joint SuperComputer Center RAS which is now the most powerful supercomputer in Russia for civil applications with the peak performance 95.04 TFlops. It has 7920 processing element cores running at 3 GHz.

4 Numerical Experiment: 3D Lid Driven Cavity

The laminar flows in the 2D lid-driven cavity is a classical test problem which is well studied both experimentally and numerically. More a challenge is studying flows in the 3D cavity (Fig. 2). We apply the algorithm described above for solving this task.

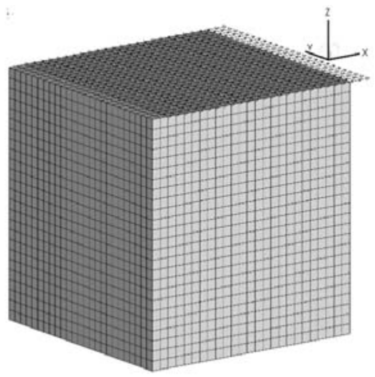


Fig. 2. General view of a 3D lid driven cubic cavity

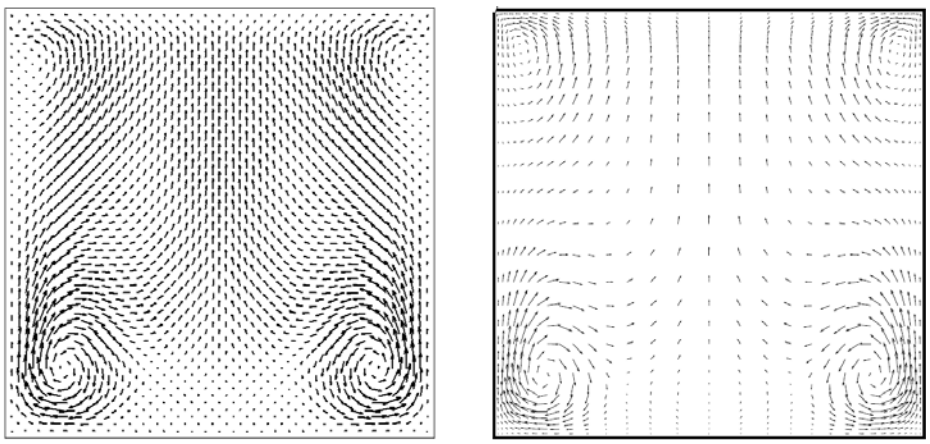


Fig. 3. Comparison of velocity fields in a vertical center-plane x = const. Left: this work; right: from [30].

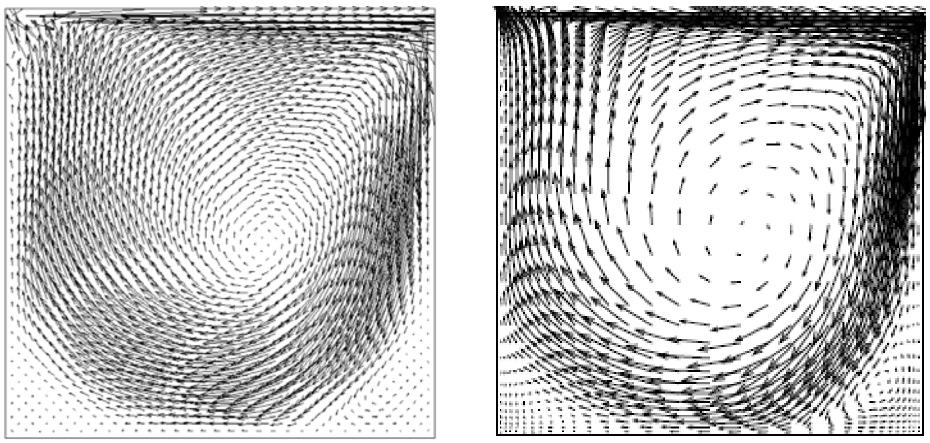


Fig. 4. Comparison of velocity fields in a vertical center-plane y = const. Left: this work; right: from [30].

Flow at a Reynolds number $\mathrm{Re} = U_L H/\nu$ of 1000 is considered here. Values of edge size $H = 1$ m and kinematic viscosity of fluid $\nu = 1.5 \cdot 10^{-5}$ m^2/s (as for air) were used. Since Re = 1000 then velocity of the lid is $U_L = 1.5 \cdot 10^{-2}$ m/s. No-slip conditions were applied on the rest walls.

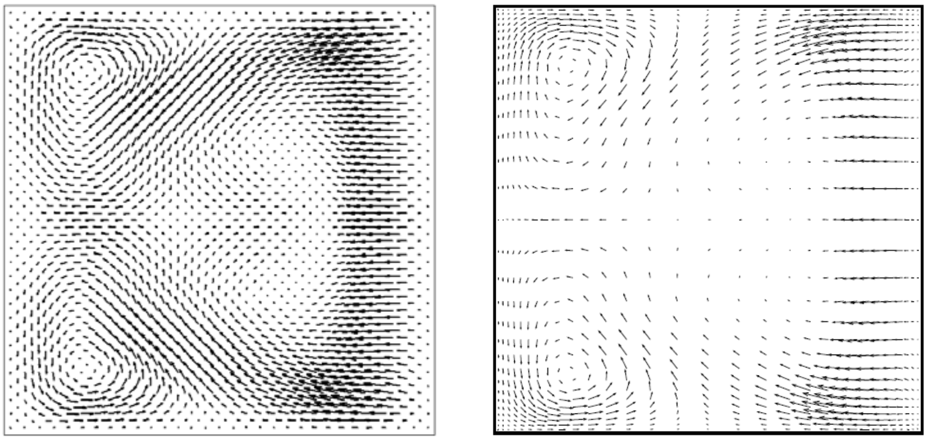


Fig. 5. Comparison of velocity fields in a horizontal center-plane $z = \text{const}$. Left: this work; right: from [30].

In Figs. 3–5 comparisons of our calculations and results of other authors [30], which used mixed boundary and finite element method, are shown. Pictures of flows in the middle sections of the cavity turned out practically identical.

For parallel calculations the cavity is split into some blocks, so that in each block there is whenever possible an equal number of nodes of the spatial grid. Each block is solved on its own processor.

Below for specification of such blocks we write $[N_x, N_y, N_z]$ where N_x, N_y, N_z are numbers of partition intervals along axes X, Y, Z, respectively. It is obvious that $N_{cpu} = N_x \cdot N_y \cdot N_z$. In particular, the record [1,1,1] means that one processor is used only.

Calculations were performed for five uniform grids, from $64 \times 64 \times 64$ to $1024 \times 1024 \times 1024$, and so that the number of nodes on each following grid increases in 8 times. In Table 1 execution time for different grids at performance of 1000 time steps and at different number of processors N_{cpu} is presented. It is visible that computation time is practically proportional to the grid size.

It is interesting also to compare execution time and efficiency for the same number of processors but with different structure of blocks.

For example, let us consider 128 processors but with different allocation on axes: [8,4,4], [4,8,4] and [4,4,8]. As shown from the Table the block [8,4,4] works usually more fast and effectively than [4,8,4] and [4,4,8]. The last block yields the worst results. It is possible to explain by accepted arrangement of Fortran-arrays in memory (the first index varies faster). For the block [8,4,4] the time for communications between processors is less than for [4,8,4] and else less than for [4,4,8]. It confirms also by statistics given out by DVM-system.

Table 1. Execution time and efficiency vs. a number of processors and structure of blocks $[N_x, N_y, N_z]$ for different grid size

No.	Number of processors	$[N_x, N_y, N_z]$	Execution time (minutes) for 1000 time steps	Eff. Coef.
	1) Grid size $64 \times 64 \times 64$			
1	1	[1,1,1]	1.437	1.000
2	2	[2,1,1]	0.900	0.960
3	2	[1,2,1]	0.974	0.947
4	2	[1,1,2]	1.021	0.936
5	4	[2,2,1]	0.807	0.913
6	8	[2,2,2]	0.846	0.686
	2) Grid size $128 \times 128 \times 128$			
7	1	[1,1,1]	15.02	1.000
8	2	[2,1,1]	8.606	0.980
9	2	[1,2,1]	8.783	0.980
10	2	[1,1,2]	8.992	0.981
11	4	[2,2,1]	6.701	0.939
12	8	[2,2,2]	7.857	0.944
	3) Grid size $256 \times 256 \times 256$			
13	32	[4,4,2]	15.48	0.850
14	64	[4,4,4]	7.132	0.648
15	128	[8,4,4]	3.601	0.613
16	128	[4,8,4]	4.047	0.490
17	128	[4,4,8]	4.167	0.451
18	256	[8,8,4]	1.859	0.496
	4) Grid size $512 \times 512 \times 512$			
19	64	[4,4,4]	61.16	0.748
20	128	[8,4,4]	31.67	0.917
21	128	[4,8,4]	31.92	0.639
22	128	[4,4,8]	34.16	0.578
	5) Grid size $1024 \times 1024 \times 1024$			
23	512	[8,8,8]	63.45	0.813
24	1024	[16,8,8]	32.83	0.902
25	1024	[8,16,8]	32.96	0.886
26	1024	[8,8,16]	47.38	0.494

5 Conclusion

The lattice Boltzmann method is a relatively new and promising numerical technique for simulating fluid flows. Discrete Boltzmann equations can be solved locally and explicitly. This method is ideal for parallel large scale computations. Fortran-DVM allows easily to create and implement debugging of parallel programs. The 3D lid-driven cavity flow is simulated by parallel LB-method. Results were compared visually with those of other authors. A good agreement is shown.

Acknowledgements

This work is supported by the Russian Foundation for Basic Research, project No. 08-01-00621.

References

1. McNamara, G.R., Zanetti, G.: Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. Phys. Rev. Lett. 61, 2332–2335 (1988)
2. Wolf-Gladrow, D.A.: Lattice-Gas Cellular Automata and Lattice Boltzmann Models. Lecture Notes in Mathematics, vol. 1725. Springer, Heidelberg (2000)
3. Succi, S.: The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. Oxford University Press, New York (2001)
4. Sukop, M.C., Thorne Jr., D.T.: Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers. Springer, Berlin (2007)
5. Qian, Y.H., D'Humieres, D., Lallemand, P.: Lattice BGK Models for Navier-Stokes Equation. Europhys. Lett. 17, 479–484 (1992)
6. Xiaoyi, H., Luo, L.-S.: Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. Phys. Rev. E. 56, 6333–6336 (1997)
7. Chen, S., Doolen, G.D.: Lattice Boltzmann methods for fluid flows. Annu. Rev. Fluid Mech. 30, 329–364 (1998)
8. Zou, Q., He, X.: On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. Phys Fluids 9, 1591–1598 (1997)
9. Brownlee, R.A., Gorban, A.N., Levesley, J.: Stability and stabilization of the lattice Boltzmann method. Phys. Rev. E. 75, 1–17 (2007)
10. Encyclopedia of Microfluidics and Nanofluidics. In: Dongqing, L. (ed.), 2226 p, in 3 volumes. Springer (2008)
11. Chen, Y.S., Shan, X.W., Chen, H.D.: New direction of computational fluid dynamics and its applications in industry. Sci. China Ser. E-Tech. Sci. 50, 521–533 (2007)
12. Hu, S., Yah, G., Shi, W.: A lattice Boltzmann model for compressible perfect gas. Acta Mechanica Sinica 13, 218–226 (1997)
13. Bing, H., Feng, W.-B., Z. Wu, Cheng, Y.-M.: Parallel Simulation of Compressible Fluid Dynamics Using Lattice Boltzmann Method. In: The First International Symposium on Optimization and Systems Biology (OSB 2007), Beijing, China, pp. 451–458 (2007)
14. Baoming, L., Kwok Daniel, Y.: A Lattice Boltzmann model with high Reynolds number in the presence of external forces to describe microfluidics. Heat and Mass Transfer 40, 843–851 (2004)
15. Zhou, Y., Zhang, R., Staroselsky, I., Chen, H.: Numerical simulation of laminar and turbulent buoyancy-driven flows using a lattice Boltzmann based algorithm. Int. J. Heat Mass Tran. 47, 4869–4879 (2004)
16. Thürey, N., Rüde, U.: Stable free surface flows with the lattice Boltzman method on adaptively coarsened grids. Computing and Visualization in Science, 179–196 (2008), doi:10.1007/s00791-008-0090-4
17. Zhou, J.G.: A lattice Boltzmann model for the shallow water equations with turbulence modeling. Int. J.l of Modern Physics C 13, 1135–1150 (2002)
18. Zhang, X., Bengough, A.G., Crawford, J.W., Young, I.M.: A lattice BGK model for advection and anisotropic dispersion equation. Advances in water Resources 25, 1–8 (2002)

19. Shiyi, C., Hudong, C., Daniel, M., William, M.: Lattice Boltzmann model for simulation of magnetohydrodynamics. Phys. Rev. Lett. 67, 3776–3779 (1991)
20. Derksen, J.J.: The Lattice-Boltzmann Method for Multiphase Fluid Flow Simulations and Euler-Lagrange Large-Eddy Simulations. In: Marchisio, D.L., Fox, R.O. (eds.) Multiphase Reacting Flows: Modelling and Simulation, pp. 181–228. Springer, Vienna (2007)
21. Swift Michael, R., Osborn, W.R., Yeomans, J.M.: Lattice Boltzmann Simulation of Nonideal Fluids. Physical Review Lett. 75, 830–833 (1995)
22. Xu, Y.-s., Liu, C.-q., Yu, H.-d.: New studying of lattice Boltzmann method for two-phase driven in porous media. Appl. Math. and Mech. 23, 387–393 (2002)
23. Shu, C., Niu, X.D., Chew, Y.T.: A Lattice Boltzmann Kinetic Model for Microflow and Heat Transfer. J. Stat. Phys. 121, 239–255 (2005)
24. Medvedev, D.A., Ershov, A.P., Kupershtokh, A.L.: Numerical Investigation of Hydrodynamic and Electrohydrodynamic Instabilities (in Russian). Dynamics of Continuous media 120, 93–103 (2002)
25. Medvedev, D.A., Kupershtokh, A.L.: Mesoscopic Simulations of Electrohydrodynamic Flows (in Russian). Fizicheskaja mezomekhanika (Physical mesomechanics) 9, 27–35 (2006)
26. Bhatnagar, P., Gross, E.P., Krook, M.K.: A model for collision processes in gases: I. small amplitude processes in charged and neutral one-component system. Phys. Rev. 94, 511–525 (1954)
27. Konovalov, N.A., Krukov, V.A., Mihailov, S.N., Pogrebtsov, A.A.: Fortran DVM — a Language for Portable Parallel Program Development. In: Proc. of Software For Multiprocessors & Supercomputers: Theory, Practice, Experience, Institute for System Programming RAS, Moscow, pp. 124–133 (1994)
28. Krukov, V.A.: Working out of Parallel Programs for Computing Clusters and Networks (in Russian). The Information Technology and Computing Systems (1-2), 42–61 (2003)
29. DVM-System, `http://www.keldysh.ru/dvm`
30. Žunič, Z., Hriberšek, M., Škerget, L., Ravnik, J.: 3D Lid Driven Cavity Flow By Mixed Boundary and Finite Element Method. In: Wesseling, P., Oñate, E., Périaux, J. (eds.) ECCOMAS CFD 2006, TU Delft, The Netherlands, pp. 1–12 (2006)

Parallel Discrete Event Simulation with AnyLogic

Mikhail Kondratyev and Maxim Garifullin

St. Petersburg State Polytechnical University,
Distributed Computing and Networking Department,
21, Politekhnicheskaya ul., St. Petersburg, Russia, 194021
Mikhail.A.Kondratyev@gmail.com, Maxim@xjtek.com

Abstract. Nowadays simulation modeling is applied for solving a wide range of problems. There are simulations which require significant performance and time resources. To decrease overall simulation time a model can be converted to a distributed system and executed on a computer network. The goal of this project is to create a library enabling clear and rapid development parallel discrete event models in AnyLogic. The library is aimed for professionals in computer simulation and helps to reduce code amount. The project includes a research on different synchronization algorithms. In this paper we present techniques which can be used in creating distributed models. We present comparison of a single threaded model with a distributed model implementing optimistic algorithm. The comparison shows a significant improvement in wallclock time achieved by separating the model into independent submodels with minimal communications.

Keywords: AnyLogic, parallel discrete event simulation, Java RMI, agent based simulation, epidemic, Time Warp.

1 Introduction

Simulation modeling nowadays is applied for a wide range of problems. Computer simulation modeling can be considered as an imitation of a real life implemented by a computer program. There are many tools for designing and executing simulation models. Some of these tools are aimed for specific application areas, some of them are general purpose simulation tools. Regardless of the tool any execution or simulation of a model requires CPU time and consumes memory. At the moment the need for global optimization causes requirements for flexibility, accuracy and level of details of simulation models to grow rapidly. These requirements often hit limitations of modern computers. For such simulations result cannot be obtained in a reasonable amount of time and therefore special techniques should be applied to decrease the wallclock time. One of the approaches is to remove certain details, events or processes from a model so that the wallclock time obviously decreases, but accuracy of output results decreases as well. If decreasing the precision of the result is impossible then distribution of a model execution can be applied.

If a computer model is distributed over a network, components of the model are executed on independent computers. Theoretically, if the communication between the submodels is negligible then increase in the execution performance is proportional to the number of computers in the network. In case of a data exchange between the model components the communication time should be added to the overall wallclock time.

Most of simulation tools which are practically used for solving problems are single threaded. Arena, Extend, VenSim are single threaded applications. To create a distributed simulation a modeler has to leave an environment to which he used to and has to recreate the model using environment which supports distributed applications. Recreation of a model may require translation of the system behavior from a modeling language to a programming language. E.g. a process description with a sequence of library blocks should be translated into Java code. Additional and really significant amount of coding should be made up to carry out support of communications between the distributed components. For professionals in computer simulation these tasks may occur to be out of their scope.

In this project we are developing techniques for rapid development of distributed simulation models. We are aimed to eliminate coding related steps of the model creating by adding library blocks directly into a simulation modeling tool. Modeler will be focused on a modeling process rather than programming by using these blocks. The modeler would need to logically divide a model into blocks with minimal communications between these blocks. Then by adding several library blocks and making a simplified setup the single threaded model will be easily turned into a distributed model. Then such a model can be executed on the computer network.

2 Simulation Platform

Most of simulation tools were originally designed for specific applications: manufacturing, material handling, financial simulations, logistics optimization. There are several platforms positioned as a general purpose tools which allow user to create a simulation of almost any real life object. In this project we are developing a solution for wide range of applications and therefore the number one criterion for selecting a simulation tool is that the chosen tool has to be a general purpose tool.

The second priority criterion is flexibility of a tool. Developing a library for distributed modeling obviously requires ability to create user libraries. Furthermore there should be enough functions and language constructs for programming complex library blocks with network interfaces. In many tools user is limited with a simplified scripting language while developing library blocks.

At the very initial stage of the project we made a research on the simulation tools using two criteria described earlier (general purpose tool and flexible library development). As the result of this research we have selected AnyLogic developed by XJ Technologies. AnyLogic is based on Java and is positioned as a general purpose simulation tool.

After the tool selection we made a review of existing Java technologies helping developing distributed applications. Any distributed application is a set of program components communicating with each other. Any single component is an independent program block, executed in a separate thread. In general when developing a distributed system programmers typically reuse one of existing platforms for messaging or data exchange over the network. Developing a customized messaging platform from scratch significantly increases a development time, leverages need for testing and later support. Using existing software or predefined components for distributed applications on a base level minimizes risks in this part of the system.

There are three widely used technologies helping programmers to create distributed applications with Java. Abbreviations are RMI, CORBA and DCOM. In this project we use RMI (figure 1).

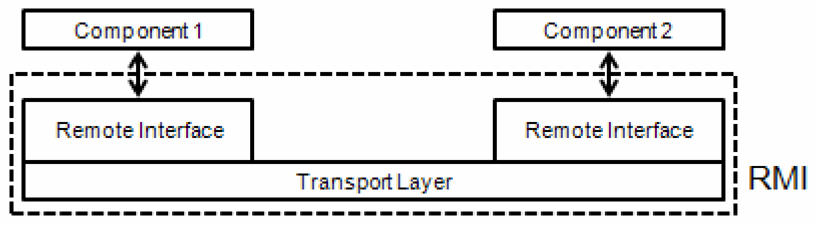


Fig. 1. Java RMI technology

The main factor influenced on our decision is that RMI is fully integrated with Java from version 1.1. AnyLogic is based on Java and allows a user to fully use power of Java in any place of a model or a library. Current version of AnyLogic uses Java 1.6 and therefore RMI is available in all versions of AnyLogic (educational, regular, older builds). An object which supposed to be accessed remotely is visible to a Java namespace via special interface. This interface should be published in RMI namespace. This interface contains description of methods which can be invoked for the object. A client interacts with the object using a reference to the published interface. Calling remote methods is clear for a programmer and is done in the same way as calling regular Java local functions. The clearness is another advantage of using RMI. RMI also supports data sending from one object to another remote object. Data is serialized using standard Java Serialization mechanisms, and then sent over the network.

The essential of a distributed simulation is a logic that synchronizes distributed parts of the model. The logic is build on top of the messaging platform and can be considered as a higher level in the overall system. The synchronization logic uses subset of RMI functions to send messages between the model components.

3 Synchronizing Model Components

AnyLogic is mostly used for creating discrete event models. In this project we will consider only discrete event simulations. Distributing continues simulations requires extremely specialized techniques and is out of scope of this project.

AnyLogic is a general purpose simulation tool which supports all of the three well known approaches for building simulation models. AnyLogic supports System Dynamics which typically deals with continuous systems with equations. Also the tool supports Agent Based and Process-Centric (which is frequently called simply the Discrete Event and utilized for queuing systems modeling) approaches used for imitating systems with discrete events. Presented library allows distribute any discrete event model, agent based or process-centric for instance. Since agent based models are the most exigent for computing resources, we considered that the library would almost be used for distributing agent based models.

The base concept of discrete event simulation is an event. Event is something that occurs in a moment of time, event is an atomic execution of a sequence of commands. While developing any discrete event model we transfer only discrete events into the model from the real world. We assume that nothing changes in between two events. Only events may cause changes in the system, also events may generate or cancel other events.

A model reflects a real life object with a set of state variables, where each variable corresponds to a property of an object. Set of values of variables characterizes a state of the system in the particular moment in a model time and the list of planned events, which should occur in model time future. In most cases of discrete event modeling the order of concurrent events execution is important. If an execution order differs from an order assumed by a model developer, results of the simulation won't be correct. Besides, an engine should ensure reproducibility of simulations runs. It means that consequent executions with the same input should produce the same result.

Event management in AnyLogic is implemented with a single event queue. This queue stores current events, which would occur in the current moment of model time, past or already processed events and future events. AnyLogic engine selects one of events scheduled for a current moment of model time. This event is then executed – a sequence of Java commands associated with this event is executed. This event may create a new event in a future or cancel an event already scheduled. Then the engine selects the next event from the events scheduled for the current moment of model time.

In case of single threaded models AnyLogic engine manages events to ensure correct sequence of events execution, but in case of parallel (distributed) discrete event simulation we divide a problem into a set of sub problems and each of them is represented with an independent AnyLogic model. Thus each submodel has its own model time and own event queue. One submodel can interact with another submodel by passing messages. As a result of receiving a message a new event will be generated in the submodel. If an event should be generated at time t, whereas the model time has progressed up to $t + \Delta t$, difficulties may take place. E.g. a plane arrives from a remote city to a local city on Jan 1st, but the local city time is Jan 10th. This event delay is called "causality problem".

To avoid causality problem one have to implement special synchronization algorithms. There are two common classes of such algorithms: conservative and optimistic algorithms [1, 2, 3].

The aim of conservative algorithms is to avoid causality problem. The basic assumption of conservative algorithms is that distributed components use FIFO channels for message passing. Thus the submodel receives messages in the same order as the remote submodel sends them. If a submodel at the moment of time t receives a message which was send by a remote submodel at time $t + \Delta t$, then there will be no messages in Δt interval coming from that remote submodel. Therefore the receiving submodel can safely continue model execution in the Δt interval assuming that there will be no new events and therefore there will be no causality problems. If there are no messages coming from remote model then computations should be paused to prevent causality problems. Upon receiving a message computations can be resumed.

The major disadvantage of conservative algorithms is low performance caused by pausing computations while waiting for an incoming message. This significantly decreases effect of running model parts in parallel. Besides that, a system running under conservative algorithm may run into a deadlock. If there is a cycle of submodels where each submodel waits for a message from each other a deadlock occurs and the whole model execution stops. Deadlock prevention or resolving requires additional logic.

Optimistic or aggressive algorithms are based on a concept of a rollback. Whereas conservative algorithms prevent system components from a causality problem, optimistic algorithms ignore such risk until it actually happens. If a causality problem is detected then optimistic algorithms use some kind of compensation algorithms. Mostly wide used algorithms are periodically saving a model state. If a causality problem occurs then the algorithm performs the model rollback to a historical state. Then a new event is inserted into a model and the model resumes its execution.

The disadvantage of optimistic algorithms is follows: when a submodel performs a rollback it should cancel all messages sent to other submodels during the rollback period. Cancellation is done via so called antimessages. Each antimessage may cause rollback in other submodels which turns into an escalating number of rollbacks all over the system. Each rollback requires CPU time and therefore a large amount of rollbacks may take greater losses than the total increase in performance caused by distributing.

There are several techniques that allow decreasing a number of rollbacks. One of the techniques is so called relaxed synchronization. This technique can be applied if order of execution of events is not very important. In this case it is assumed that all events in a certain time window are imminent and therefore an order of execution is not important. Using this technique may violate the requirement of reproducible results, but significantly increases performance of optimistic algorithms.

Conservative and optimistic algorithms are two mostly used algorithms for synchronizing distributed submodels. Depending on a problem each algorithm will give different performance gain. In a distributed system the algorithm is the number one factor which influences the success of turning a single threaded model into a distributed one.

4 Example of a Distributed Agent Based Model

In this project we are developing a library that enables a simulation modeler to extend a model with an ability to run on a computer network. The model parts can be synchronized using different algorithms therefore allowing a modeler to choose an optimal synchronization algorithm.

The first version of the library is designed to support a subclass of discrete event models. The subclass is called Agent Based modeling and is a type of modeling which is widely used to solve many different problems. There is no common definition of what is agent based and what is agent. People still discuss what properties should a thing have to be called an agent. Commonly referenced features are ability to move, ability to learn, communications etc. From practical point of view we will stress only one feature common for all agent based models. Agents based models are essentially decentralized. There is no such place where a global behavior of a system is defined. On a contrary, a modeler defines a system behavior on a level of individual components and the global behavior emerges as the result of many individuals sharing the same environment, following their own local rules and communicating with each other. This approach is widely used for describing complex social and economic models. Besides flexibility agent based models typically require much more computational resources than traditional equation based models. There are simulations where accuracy can be achieved only by modeling millions of agents concurrently in the same environment with the same timescale. Therefore we expect that this project will give a lot of benefits and will break certain limits in agent based simulations.

Up to this moment we have developed as baseline version of the library which is a part of AnyLogic simulation tool. The library uses RMI for passing messages between distributed parts. In the following section an example of creating a distributed model is provided. The example is based on a classical agent based problem of an epidemic.

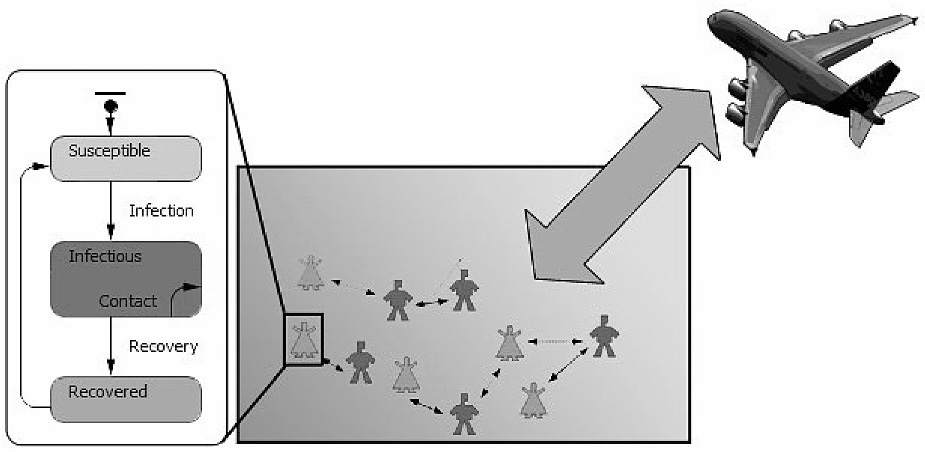


Fig. 2. Model of behavior of agents in particular city. On the left there is a statechart representing the behavior of each agent.

The problem considers a region with cities. People live in cities, communicate with each other and travel between the cities. The assumption is that people or agents leave in metric space and can be Susceptible, then Infectious if infected with another person and than being Recovered. Disease spreading is distance based – only in a certain range infected person may pass a disease to a susceptible person (figure 2).

If a simulation consists of a single city then simulation obviously can be done in a reasonable amount of time. Simulation complexity exponentially grows with a number of agents and for millions of agents in metric space a simulation often hits performance limitations. For example Moscow region in Russia contains more than 10 millions of people and simulation of this population is computationally expensive.

Even more important problem is getting accurate prediction of the disease spreading on a country basis. With several highly populated regions modeling the whole country is an extremely expensive in terms of computation resources. Keeping results accurate prevents from scaling the country down or using techniques like grouping agents. Then distributed modeling can be applied to decrease the overall simulation time. Converting the model into a distributed system is nontrivial task because there are communications between people and movements of people between cities (figure 3).

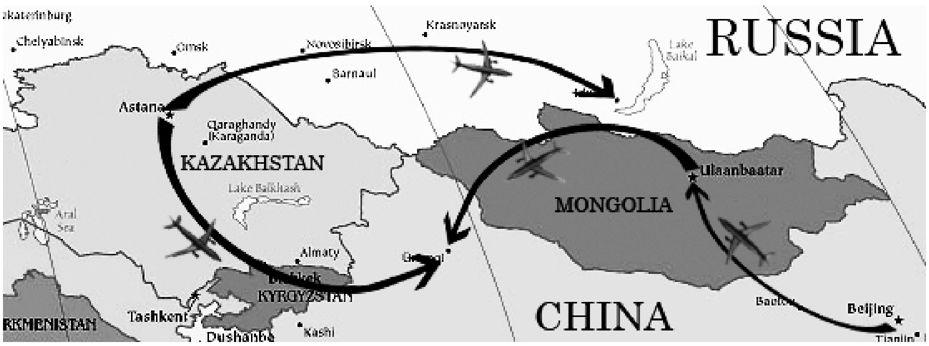


Fig. 3. Migration of agents

In this example problem we represent movements between cities as periodic flights with a certain number of people onboard. There are no predefined schedules for flights, just randomly selected moments of time.

This model was developed in AnyLogic in several versions. The first model was designed as a single threaded model and was used for later comparison. The model includes three cities. This baseline model can be further extended with additional agent behaviors like family contacts, employment and etc. In this case study we were analyzing spreading of a disease with one agent initially infected in a certain city.

The model was distributed into three parts each containing a single city. Each part was executed in a separate thread. Planes were represented as messages with parameters, i.e. number of passengers onboard. For message passing we use RMI (a technology for rapid development of distributed systems integrated with Java 1.1). Submodels are synchronized using Time Warp algorithm [4].

Defining a remotely visible object starts from defining a special Java interface which should extend a predefined interface *java.rmi.Remote*. This interface defines methods which can be accessed remotely. Methods can be extended to receive parameters – either of primitive types (integer, double, logical values etc.) or compound types. In case of using the compound types each type must be serializable. In this case Java creates a copy of such variable and sends the copy to the network.

After defining the remote interface the interface should be implemented by a class. The class must extend *java.rmi.server.UnicastRemoteObject*. The last step is to create a skeleton and a stub using *rmic* compiler which is provided in Java Development Kit. Stubs and skeletons are used while making remote calls (figure 4).

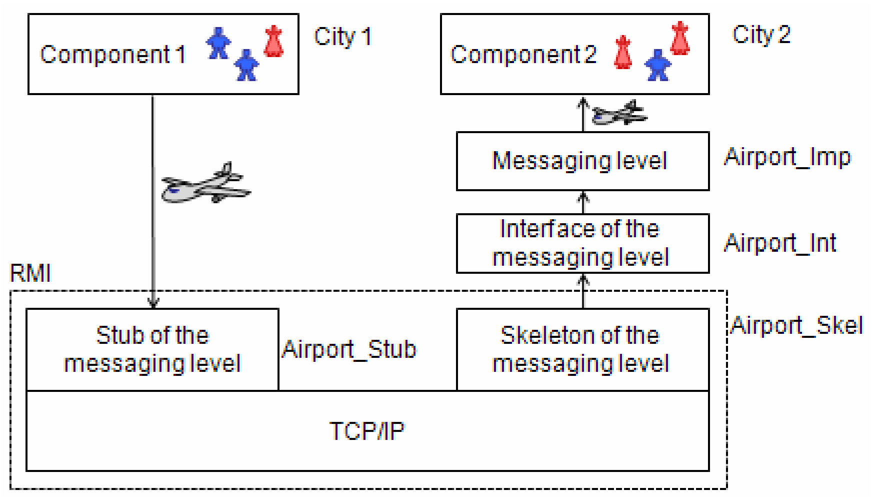


Fig. 4. Java RMI technology in action. Stubs and skeletons are used while making remote calls.

To send a message (plane with passengers) from a component 1 to a component 2 the component 1 obtains a remote reference to an object Airport_Imp. The reference is obtained using a special RMI naming service which resolves string names into references. The object Airport_Imp implements functions for message passing and can be considered as an intermediate level between the simulation model and RMI scope. Functions of the object Airport_Imp are defined in a remote interface Airport_Int and therefore are available remotely.

Instead of direct accessing the methods of Airport_Imp (e.g. accept a plain) the component 1 invokes methods of a stub Airport_Stub. The stub is a lightweight copy of Airport_Imp and is the only one class needed for the component 1 to be able to access Airport_Imp object. Implementation and the component 2 are located on a separate computer.

RMI transmits the function call from the stub into a skeleton Airport_Skel over TCP/IP. The skeleton is created on the component 2 side. The skeleton receives parameters values and invokes the corresponding function of Airport_Imp. This invocation may pass a message to the model and the model may return data into the component 1.

In the following section we present performance analysis for the model. Figure 5 shows a dependency of the overall wallclock time from a number of agents. Agents were uniformly distributed over all cities. The diagram contains two curves – one is for the single threaded model and another one is for the distributed version with three threads. The diagram shows 7-9 times better performance for the distributed version. One of the reasons of this performance improvement is a type of updates used in the single threaded model by the AnyLogic. When some agents interact with each other all other agents are refreshed. If a single agents gets sick then agents in all other cities were updated which never happens in a real life.

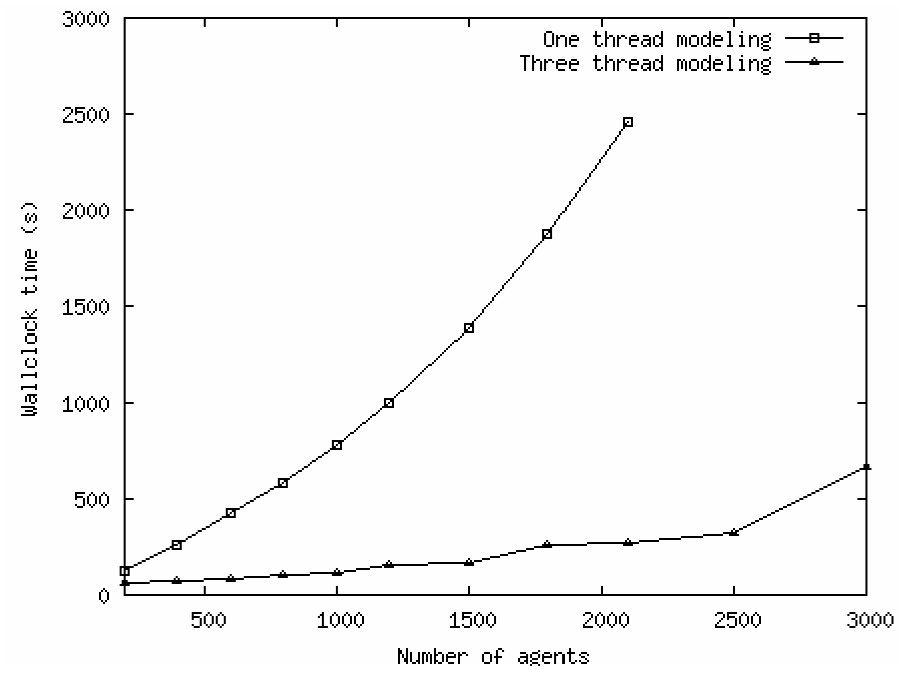


Fig. 5. Dependence of wallclock time on the number of agents

All these experiments were done on a Cure 2 Duo processor. Therefore one of the threads was executed on a single core and two others were sharing the second core. We expect even better results if using 3 independent computers or a processor with more than 2 cores.

The model has proven the effect of converting the single threaded model into a distributed model. The simulation time was decreased significantly using optimistic algorithm with rollbacks. In certain other simulation using purely optimistic algorithms potentially may give the same time as the single threaded model. Optimistic algorithms periodically save a system state including all agent states into a memory. In this particular model we were saving the system state each time a plane departs. Increasing a size of agents population will increase memory consumption and will definitely increase the time needed to save a state and later reload the state if a rollback is needed. Using distributed approach gives performance improvements over single threaded models event in this case, but after a certain point optimistic algorithms should be replaced with conservative algorithms.

Generally speaking, models with heavy computations and relatively small memory requirements can be efficiently synchronized with optimistic algorithms. Conservative algorithms are capable for simulation models with greater memory requirements and relatively simple event processing.

Another factor influencing significant performance improvements in this model is tuning synchronization algorithms with additional information from the problem. E.g. we have considered average trip time between the cities. If a plane departs at time *t*, then arriving of the plane is at time $t + \Delta t$. Considering Δt while saving states prevents algorithms from doing unnecessary rollbacks. Figure 6 shows a dependency of an overall simulation time from the value of Δt. The assumption of the model was that distance and therefore the travel time is the same for all three cities. Decrease in the simulation time is caused not only by increase in algorithms efficiency, but also because of fewer events in the model. With the fixed number of planes the greater travel time decreases number of departures and arrivals in the same period of time.

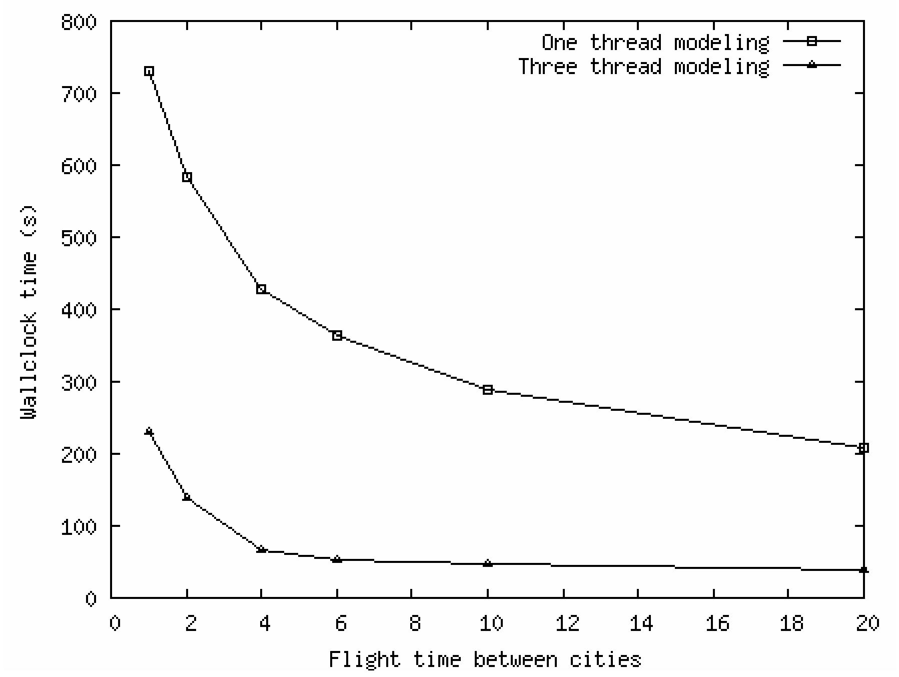


Fig. 6. Dependence of wallclock time on plane travel time

5 Conclusion

The example model has proven the efficiency of converting a single threaded model into a distributed model. These results can be further improved, by using smarter algorithms with a relaxed synchronization. If a plane arrives to a city at 1:00 p.m. but the city time is 1:10 p.m. then causality problem occurs. If the time difference is negligible then a model can skip a rollback and process the plane at 1:10 p.m. For this particular application this approach has an intuitive connection with a real life when a plane is delayed and arrives a bit later than scheduled. Ignoring the gap and processing the plane later allows avoiding spending CPU time on the rollback.

For certain agent based models using optimistic algorithms and saving a model state only while sending messages can be less effective. E.g. there can be fairly small amount of messages between submodels and heavy calculation in between. If a roll-back occurs then a significant amount of calculation is lost. In this case the submodel can be extended with an ability to save its state periodically between sending messages to other submodels.

We plan to extend the model with ability to switch between synchronization algorithms. The basic version of the algorithms will be extended with additional smart elements, e.g. relaxed synchronization. So far we have implemented a part of the library blocks. Each block has a graphical interface and can be used for clear conversion of a single threaded AnyLogic model into a distributed AnyLogic model. The library allows selecting either optimistic or conservative algorithm.

The library significantly increases performance not only when running a model on a network of computers, but also when running a model on multi core processors.

References

1. Fujimoto, R.M.: Parallel discrete event simulation. In: Proc. of the Winter Simulation Conf., pp. 19–28 (1989)
2. Fujimoto, R.M.: Distributed simulation systems. In: Proc. of the Winter Simulation Conf., pp. 124–134 (2003)
3. Perumalla, K.S.: Parallel and distributed simulation: traditional techniques and recent advances. In: Proc. of the Winter Simulation Conf., pp. 84–95 (2006)
4. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems 7(3), 404–425 (1985)

LGA Method for 1D Sound Wave Simulation in Inhomogeneous Media*

Valentina Markova

Supercomputer Software Department
ICM&MG, Siberian Branch, Russian Academy of Sciences
Pr. Lavrentieva, 6, Novosibirsk, 630090, Russia
markova@ssd.sscc.ru

Abstract. The Lattice Gas Automata (LGA) models are based on a microscopic model of physical process and can be considered as an adjunct to the traditional numerical methods to the spatial dynamics simulation. Here we consider two simple LGA models (HPPrp and HPP). They are based on a regular two-dimensional four-neighbors Euclidean lattice. Lattice nodes can be occupied by the moving particles and the moving. In this paper, the possibility of the LGA models to simulate sound wave process in inhomogeneous media formed from two gases (helium and methane) are investigated.

1 Introduction

In the Lattice Gag Automata (LGA) models [1-3], dynamics of an event is described by a set of hypothetical particles, which have moved through space and collided with each other and with obstacles. The space is represented as a regular lattice whose nodes can contain a quantity of hypothetical particles. Each lattice node is assigned to a LGA cell. As opposed to the classical cellular automaton, an initial state of the LGA cell is determined by a set of some particles, locating in the cell at this time moment. There are two types of particles: the moving particles and the rest particles. The moving particles have unit mass and unit velocity. The rest particles have the same velocity (equal to zero) and a different mass. Interactions between particles are simple. Each interaction consists of two successive steps: collision and propagation. The collision rules are chosen in such a way that the mass and momentum conservation laws are satisfied. The collision rules determine the LGA cell transition table. All cells update their own states simultaneously and synchronously. An iterative change of the LGA global state (evolution of the LGA) describes the dynamics of an event on microscopic level.

In this paper,the ability of a simple LGA models to simulate sound wave propagation in inhomogeneous medium by the example of a medium from two gases (helium and methane). As the models, two LGA models (HPP and HPPrp)

* Supported by 1)Presidium of Russian Academy of Sciences, Basic Research Program N 2 (2009), 2) Siberian Branch of Russian Academy of Sciences, Interdisciplinary Project N (2009).

on regular two-dimensional four-neighbor Euclidean lattice are used. The HPP model is a special case of the HPPrp model: the HPPrp cells contain not only the moving particles, but the rest particles as well. In [3] it is shown that the HPPrp model corresponds to the wave equation and in addition allows to realize media with different sound wave velocity.

This paper is organized as follows. After the Introduction, in the second section, the main concepts of the HPPrp models is given. The third section is concerned with an experimental study of 1D sound wave propagation in the HPPrp medium. Here, the influence of the HPPrp model parameters on the velocity of 1D sound wave propagation is studied. Technique for evaluation the HPPrp medium and its parameters by physical velocity of the sound wave propagation in the given medium and vice versa is suggested. A simple example of 1D sound wave propagation in inhomogeneous medium (helium and methane) is discussed.

2 HPPrp Models of 1D Wave Propagation Process

2.1 The HPPrp Model

The HPPrp model is defined on regular two-dimensional four-neighbor Euclidean lattice. Each HPPrp cell can contain the moving particles (mov) and the rest particles (rp). The moving particles have unit mass and unit speed. No more than one moving particle may occupy a given lattice site or move in a given direction at a given time. The rest particles have the same velocity (equal to zero) and a different mass. Here we will consider the rest particles with the masses equal to 2, 4, 8, and 16.

The HPPrp cell state is defined by the two vectors: the velocity vector $\overrightarrow{v}$ and the mass vector $\overrightarrow{m}$. The length of the velocity vector is equal to the number of neighbors, i.e., $\overrightarrow{v} = (v_1, v_2, v_3, v_4)$. The l-th digit value of the vector, $l = 1, 2, 3, 4$, shows the presence ($v_l = 1$) or the absence ($v_l = 0$) of the moving particles in the direction to the l-th neighbor. The length of the mass vector is equal to the number of neighbors plus the number of the rest particles $\overrightarrow{m} = (m_{4+b_r}, m_{3+b_r}, \dots, m_1)$. The l-th digit value of the vector, $l = 1, 2, \dots, 4 + b_r$, determines the presence ($m_l = 1$) or the absence ($m_l = 0$) of a rest particle with mass 2^{l-4} in a cell. So, the state of a HPPrp cell is represented by two Boolean vectors: the first vector of length 4 (the velocity vector) and the second vector of length $(4 + b_r)$ (the mass vector). (In the following, the HPPrp model with one rest particle will be indicated by HPP1rp, the HPPrp model with two rest particles will be indicated by HPP2rp and so on.) The HPP3rp cell with the velocity vector $\overrightarrow{v} = (0, 1, 1, 0)$ and the mass vector $\overrightarrow{m} = (\mathbf{0}, \mathbf{1}, \mathbf{1}, 0, 1, 1, 0)$ is shown in Figure 1a. (An arrow in the cell shows the direction of the velocity vector particle.)

2.2 The HPPrp Model Behavior

Each interaction of the HPPrp-cell consists of two successive steps: collision and propagation.

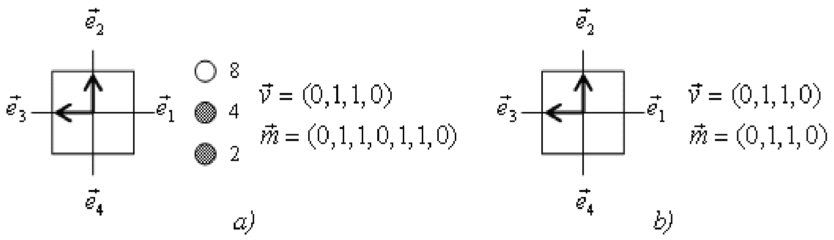


Fig. 1. Examples of cells: the HPP3rp cell (a) and HPP cell (b)

Propagation step. In the propagation step, in each cell all moving particles move in the direction defined by the bit position in the velocity vector.

Collision step. In the HPPrp model, the energy exchange may occur not only between moving particles in each cell, but between the moving and the rest particles as well. In response to this exchange, either a rest particle is created and moving particles are annihilated or a rest particle is annihilated and moving particles are created. In the general case, the collision rules are deterministic or non-deterministic. They can be divided into the three groups.

Group 1 (head-on collision). The moving particles collide with each other according to head-on collision rule (Figure 2) independent of the presence or the absence of the rest particles.

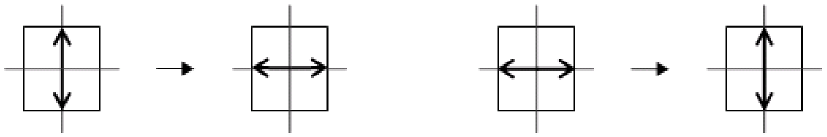


Fig. 2. Collision rules

Group 2 (rest particle creation). If in a cell, the collision rule is hold for two (four) moving particles and there is initially no mass 2 (4) rest particle, then moving particles will be annihilated and a mass 2 (4) rest particle will be created, respectively (Figure 3a). If in a cell, the collision rules are hold for two moving particles and there are initially no mass 4 rest particle and mass 2 rest particle, then moving particles and mass 2 rest particle will be annihilated and a mass 4 rest particle will be created (Figure 3b).

Group 3 (rest particle annihilation). If a mass 2 (4) rest particle already exists in the cell, and there are no two (four) moving particles for which the collision rule is hold, then two (four) moving particles will be created after the collision step, respectively, and the rest particle will be annihilated (Figure 4a). If a mass 4 rest particle already exists in the cell, and there are no a mass 2 (4)

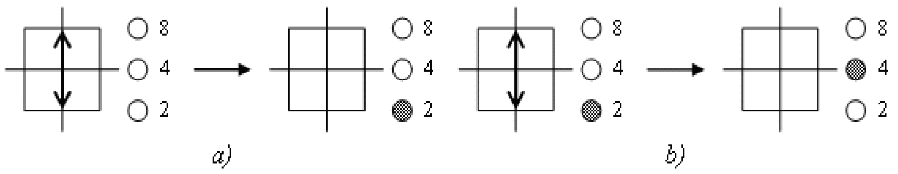


Fig. 3. The rest particle creation rules

rest particle and no four (or two) moving particles for which the collision rules are hold, then two moving particles and a mass 2 rest particle will be created after the collision step, and a mass 4 rest particle will be annihilated (Figure 4b).

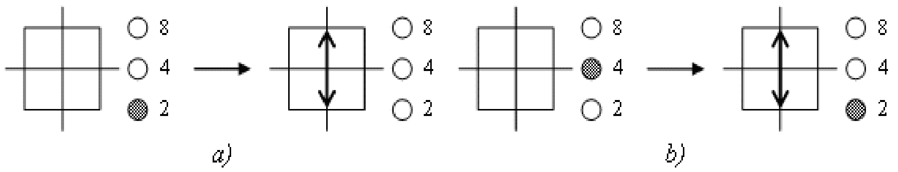


Fig. 4. The rest particle annihilation rules

The rest particles are created (annihilated) with a certain probability $\mathcal{P}_k$, $k = 1, 2, \ldots, b_r$, in so doing, the following limitations should be met

$$\mathcal{P}_{k+1} \geq \mathcal{P}_k, \qquad \sum_{k=1}^{b_r} \mathcal{P}_k \leq 1. \tag{1}$$

Unfortunately, the HPPrp models have some spurious conservation laws, such as the total momentum in the i-direction along each horizontal line and the total momentum in the j-direction along each vertical line.

So, the HPPrp models are characterized by the following parameters.

- A set of probabilities of moving particles presence in the initial particle distribution $P^{mov} = \langle p_4^{mov}, p_3^{mov}, p_2^{mov}, p_1^{mov} \rangle$.
- Number of rest particles and their mass.
- A set of probabilities of rest particles presence in the initial particle distribution $P^{rp} = \langle p_{b_r}^{rp}, p_{b_r-1}^{rp}, ..., p_1^{rp} \rangle$.
- A probabilities of creating and annihilating of the rest particles for which the conditions (1) should be met.

The HPP model is a special case of the HPPrp model. The HPP cells contain only moving particles. Collision rules are limited by head-on collision rules. he HPP model has superfluous laws of conservation: the total mass and moment are conserved along each space axis individually. In addition, the HPP model does not satisfy all the conditions of isotropy. That is the reason that the HPP model has limited usefulness in the physical modeling. In [3], as an example, a capability of simulating three-dimensional electromagnetic fields has been demonstrated. The HPP cell with the velocity vector $\overrightarrow{v} = (0,1,1,0)$ and the mass vector $\overrightarrow{m} = (0,1,1,0)$ is shown in Figure 1b.

2.3 Averaged Values

In terms of the cellular automata modeling, a 2D cellular array $W = \{w(i,j)\}$ with $M \times N$ size cells is represented a medium wherein any physical event is observed. The array cells are the HPPrp cells. A finite-state automaton is assigned to each cell. The automaton transition table has 2^{b_r+4} states. The HPPrp cells initial states are generated according to two sets: the set of probabilities of moving particles presence P^{mov} and the set of probabilities of rest particles presence P^{rp}. Collision and propagation rules make up the automaton transition rules. Further the array of the HPPrp cells will be called by *the HPPrp medium*. Each medium cell with coordinate $w(i,j)$ is given by two parameters: the velocity vector $\overrightarrow{v}_{ij} = (v_4(ij), v_3(ij), v_2(ij), v_1(ij))$ and the mass vector $\overrightarrow{m}_{ij} = (m_{4+b_r}(ij), m_{3+b_r}(ij), \ldots, m_1(ij))$.

In the simulation, the enumerated parameters of the medium cells have no practical significance. But the averaged values of this parameters over some *averaging area* $Av(ij)$ are of interest. The area $Av(ij)$ includes all cells with coordinates $(\hat{i},\hat{j})$ placed not farther from a cell with coordinate (i,j), than at a certain distance r called *the averaging radius*. In our case, the area $Av(ij)$ defines a square $(2r+1)$ cells on side. Further the medium cell with coordinate (i,j) will be given by the following averaged values

$$\langle \rho_{ij}^{mov} \rangle = \frac{1}{|Av(ij)|} \sum_{(\hat{i},\hat{j})\in Av(ij)} \sum_{l=1}^{4} m_l(\hat{i},\hat{j}),$$

$$\langle \rho_{ij}^{rp} \rangle = \frac{1}{|Av(ij)|} \sum_{(\hat{i},\hat{j})\in Av(ij)} \left(\sum_{l=1}^{4} m_l(\hat{i},\hat{j}) + \sum_{l=5}^{b_r+4} 2^{l-4} m_l(\hat{i},\hat{j}) \right), \tag{2}$$

$$\langle \overrightarrow{u}_{ij} \rangle = \frac{1}{|Av(ij)|} \sum_{(\hat{i},\hat{j})\in Av(ij)} v_1(\hat{i},\hat{j})\overrightarrow{e}_1 + v_2(\hat{i},\hat{j})\overrightarrow{e}_2 + v_3(\hat{i},\hat{j})\overrightarrow{e}_3 + v_4(\hat{i},\hat{j})\overrightarrow{e}_4,$$

where $\langle \rho_{ij}^{mov} \rangle$ is the averaged density of the moving particles, $\langle \rho_{ij}^{rp} \rangle$ is the averaged density of all particles in the HPPrp cell, $\langle \overrightarrow{u}_{ij} \rangle$ is the averaged of the velocity vector, $|Av(ij)|$ – is the number of the cells situated in $Av(ij)$. The averaged projection of the velocity vector on the axis X (Y) is convenient to be used for our purpose

$$\langle v_{ij}^{x} \rangle = \frac{1}{|Av(ij)|} \sum_{(\hat{i},\hat{j})\in Av(ij)} v_1(\hat{i},\hat{j}) - v_3(\hat{i},\hat{j}). \tag{3}$$

(For simplicity, $\langle v_{ij}^x \rangle$ will be called by *the averaged projection of the velocity vector.*) In the Table 1 averaged values of density $\langle \rho^{rp} \rangle$ for all HPPrp media in the equilibrium state are listed. It implies that the probability of all rest particle presence in initial distribution equals to 0,5. Further the density $\langle \rho^{rp} \rangle$ will de called *the model density of HPPrp medium.*

Table 1.

rp	*LGA model*	$\langle \rho^{rp} \rangle$
0	HPP	2
1	HPP1rp (m_1 = 2)	3
2	HPP2rp (m_2 = 4, m_1 = 2)	5
3	HPP3rp (m_3 = 8, m_2 = 4, m_1=2)	9
4	HPP4rp (m_4 = 16, m_3 = 8, m_2=4, m_1=2)	17

In [3] it is shown, that if a small disturbance with density ρ is superposed onto an equilibrium state of the HPPrp with density ρ^{pr} and zero velocity, then the behavior of the HPPrp medium adheres to the linear wave equation in terms of density ρ:

$$\frac{\partial^2 \rho}{\partial t^2} - v_{ph}^2 \nabla^2 \rho = 0,$$

where v_{ph} is the sound wave velocity.

3 Experimental Study of 1D Sound Wave Propagation in the HPPrp Medium

3.1 1D Sound Wave Propagation Simulation

In the experiments carried out, the propagation of 1D unit sound wave propagation process is presented by evolution as evolution of a cellular array W of size 200 × 2000 cells. A source for generation of initial momentum is located in the center of W and represents a subarray H of 200 × 100 cells. Each source cell generates several particles within one iteration. These can be either the moving particles and (or) the rest ones. Cells of the rest part of the array W are the HP-Prp cells. The initial states of the array cells were generated according to a set of probabilities of the rest particles P^{rp} and a set of probabilities the moving ones presence P^{mov} in the initial particle distribution for the given HPPrp medium and source. The boundary conditions are periodical. A radius of averaging r=25. Since the wave process is symmetric with respect to a source, we will consider only the right part of the array W in the subsequent.

The wave process in the HPPrp medium will be given by a changing two functions.

– The dependence of averaged over the array columns of $\langle \rho_{ij}^{rp} \rangle(t)$ (2) (twice averaged density of the HPPrp medium cell) on time

$$\langle \rho_j^{rp} \rangle(t) = \frac{\sum_{i=0}^{M-1} \langle \rho_{ij}^{rp} \rangle(t)}{M}, j = 0, 1, ..., N/2,$$

where $\langle \rho_{ij}^{rp} \rangle(t)$ is an averaged model density value of the medium cell with coordinate (i, j) at the time instant t.

– The dependence of averaged over the array columns of $\langle v_{ij}^x \rangle(t)$ (3) (twice averaged projection of the velocity vector of the moving particles onto the axis X) on time

$$\langle v_j^x \rangle(t) = \frac{\sum_{i=0}^{M-1} \langle v_{ij}^x \rangle(t)}{M}, j = 0, 1, ..., N/2,$$

where $\langle v_{ij}^x \rangle(t)$ is an averaged projection of the velocity vector of the medium cell with coordinate (i, j) at the the time instant t.

Let us consider a unit wave process in the HPP medium. The unperturbed HPP media is at equilibrium state $(\langle \rho_{ij}^{mov} \rangle = 2$ for all $i = 0, 1, ..., M$ and $j = 0, 1, ..., N/2$. Let each disturbance source cell generates three moving particles with equal probabilities presence in the initial particle distribution, i.e., $p_1^{mov} = p_2^{mov} = p_3^{mov} = p_4^{mov} = 1$. This means that a velocity vector of each source cell equals one of four values: $\overrightarrow{v}_{ij}^1 = (0, 1, 1, 1)$, $\overrightarrow{v}_{ij}^2 = (1, 0, 1, 1)$, $\overrightarrow{v}_{ij}^3 = (1, 1, 0, 1)$, $\overrightarrow{v}_{ij}^4 = (1, 1, 1, 0)$. The projections of the above velocity vectors onto axis OX differ in value and sign: $(v_{ij}^x)^1 = +1$ (a particle moving in the direction of wave propagation), $(v_{ij}^x)^4 = -1$ (a particle moving in the opposite direction of wave propagation), $(v_{ij}^x)^2 = (v_{ij}^x)^3 = 0$ (a moving along axis OX is absent). Hence, $\langle(v_j^x)\rangle(0) = 0$ for all $j = 0, 1, ..., N/2$. Further this source will be represented by S1. A unit wave is formed as a result of action of the disturbance source S1. The generated unit wave process is shown at Figure 5 and Figure 6. Figure 5 presents twice averaged density of the HPP medium for $t = 0$, $t = 200$, $t = 300$, $t = 900$ and $t = 1000$. Figure 6 illustrates twice averaged projection of the velocity vector of the moving particles onto the axis X for $t = 0$, $t = 200$, $t = 300$, $t = 900$ and $t = 1000$.

3.2 The Influence of Moving Particles Direction in Cells of a Source

Figure 7 presents two twice averaged projection of the velocity vector onto the axis OX for $t = 200$, corresponding two 1D unit sound waves. The waves are generated by different sources (S1 and S2). The cells of both sources are the HPP cells. They contain three moving particles, and differ by moving particles direction in the initial state. As opposite to the cells of the source S1, cells of the source S2 generates three moving particles with unequal probabilities presence in the initial particle distribution, namely, $p_1^{mov} = p_2^{mov} = p_3^{mov} = 1$, $p_4^{mov} = 0$. This means that all cells of the source S2 are the some value of velocity vector equals to $\overrightarrow{v}_{ij} = (0, 1, 1, 1)$. Hence, $(v_j^x)(0) = 1$ for all $j = 0, 1, ..., N/2$.

The experiments have shown that a maximum value of the twice averaged vector projection, corresponding to the first wave, exceeds that of the twice averaged vector projection, corresponding to the second wave. Note that the first wave ranks below the second one in the second parameter (width).

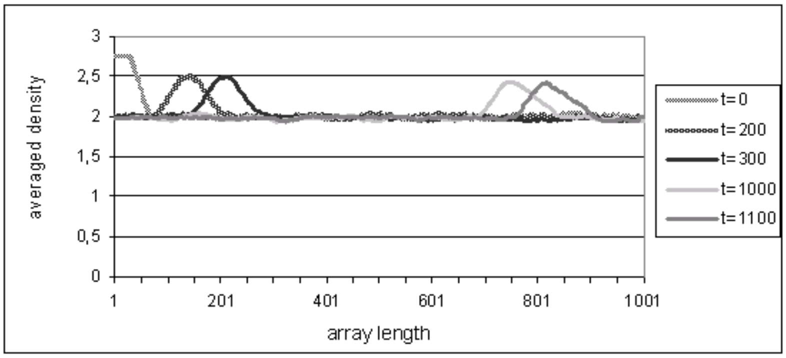


Fig. 5. The dependence of twice averaged density of the HPP medium cell on time

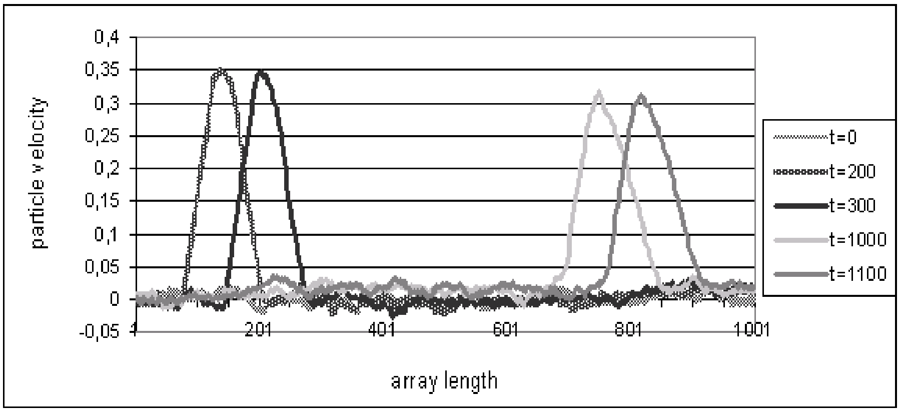


Fig. 6. The dependence of twice projection of the velocity vector onto the axis X on time

3.3 The Influence of Rest Particle Number in Cells of a Source

Figure 8 shows two twice averaged cell density in equilibrium HPP medium with two source (S2 and S3) for $t = 400$. The first source cells are the HPP cells, they contain three moving particles with unequal probabilities presence in the initial particle distribution: $p_1^{mov} = p_2^{mov} = p_3^{mov} = 1$, $p_4^{mov} = 0$. Hence, $\vec{m}_{ij} = (0, 1, 1, 1)$ for all $i = 0, 1, ..., M$ and $j = 0, 1, ..., N/2$. The second source cells are the HPP1rp cells with the following probabilities of the rest and the moving particles presence in the initial particle distribution: $p_1^{mov} = p_2^{mov} = p_3^{mov} = 1$, $p_4^{mov} = 0$ and $p_1^{rp} = 1$. This means that all cells of the source S3 are the some value of mass vector equals to $\vec{m}_{ij} = (\mathbf{1}, 0, 1, 1, 1)$. In the source S3, a mass 2 rest particles are annihilated after the collision step according to the following rules: $(\mathbf{1}, 0, 0, 0, 0) \implies (\mathbf{0}, 1, 0, 1, 0) \vee (\mathbf{0}, 0, 1, 0, 1)$; $(\mathbf{1}, 1, 0, 1, 0) \implies (\mathbf{0}, 1, 1, 1, 1)$;

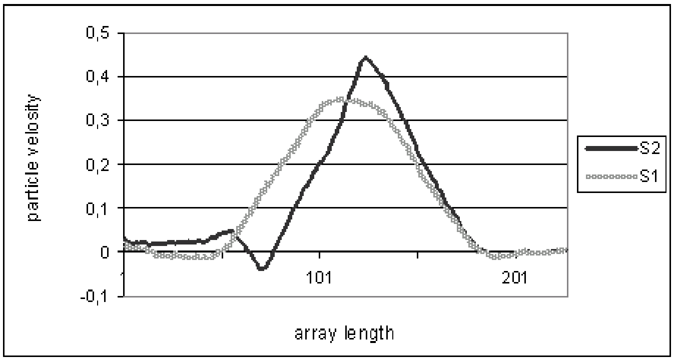


Fig. 7. Two twice averaged projection of velocity vector onto the axis OX for $t = 200$

$(\mathbf{1}, 0, 1, 0, 1) \implies (\mathbf{0}, 1, 1, 1, 1)$. As a result of moving particles creation in the cells of the source S3, the generated unit wave exceeds the one, resulted from the action of the source S3 on the HPP medium in the maximal density value. If we introduce the collision rules of rest particles in the cells of a source, then as a result a wave front will form.

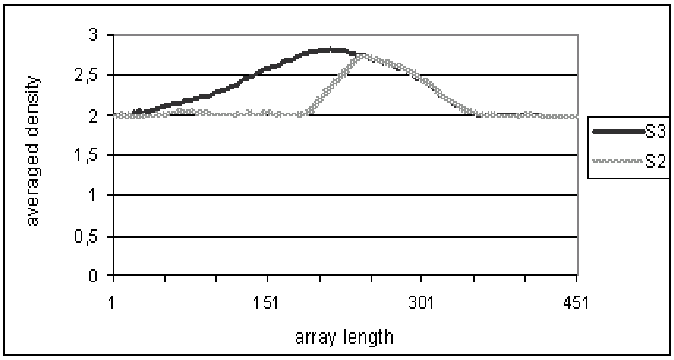


Fig. 8. Two twice averaged density of the HPP medium with two source for $t = 400$

A large body of performed experiments pointed to the fact that the density function amplitude is diminished as the iteration number increases, the wave-front ia washed out. In doing so,the mass and momentum conservation laws are satisfied. So, the HPPrp wave model captures a diffusion effect as opposed to wave equation.

3.4 Determination of Velocity of the Sound Wave Propagation

The velocity of the sound wave propagation is defined as the number of array cells for which maximum value of the model density travels in a unit time. (Further the velocity of the sound wave propagation in the HPPrp medium will de called *the model velocity* v_m). Model velocity is measured in terms cells/iteration. Since the moving particles were allowed to move only in the direction of their nearest neighbors, $| v_m |< 1$. In Figure 9 the model velocity dependence of the moving particle density for medium at the equilibrium state is shown. The velocity curves are symmetric about the density $\langle \rho^{mov} \rangle = 2$, regardless of the numbers of the rest particles and the probability of their presence. In all HPPrp media, the sound waves achieve maximum velocity and the greatest difference in magnitude for the density $\langle \rho^{m} \rangle = 2$. In [4] two methods of changing the model velocity are presented. The first method is based on change in the rest particle initial distribution, the second one is based on change in the collision rules.

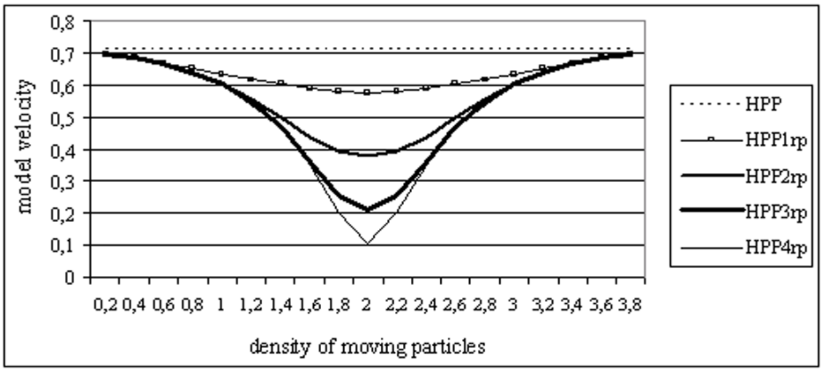


Fig. 9. The model velocity dependence of density of the moving particles

3.5 Correspondence between the Model and the Physical Sound Wave Velocity

In order for the sound wave propagation to be simulated in different physical media (solid, gas, fluid) each physical medium must be assigned the HPPrp medium by physical velocity v_{ph} with which the sound wave travels in the given physical medium and vice versa. For this purpose, a scale coefficient for velocity conversion (from physical velocity to model velocity and vice versa) must be defined. Let the sound wave propagates in gases be 965 – 313 m/s. The sound wave reaches maximum physical velocity in helium, maximum model velocity in the HPP medium. As a result, helium is assigned the HPP medium. In the general case, any HPPrp medium can be used as a basic media under the following condition: the ratio between the physical velocity of sound wave in gases do not exceed the ratio between the model velocity of sound wave in the HPPrp media corresponding to the given gases. Further the HPP medium will be considered as a basic one, then scale coefficient is defined as

$$\mu_{hel} = v_{ph}/v_m = 1,38 \cdot 10^3.$$

For example, the sound wave propagates in methane with the velocity $v_{ph} =$ 439 m/s. According to the scale coefficient μ_{hel}, the model sound velocity in methane $v_m = 0,31$. The sound wave propagates with such velocity in all HPPrp media but for different density of moving particles and the probability of the rest particles presence in the initial particle distribution (Figure 6). Which of the all HPPrp medium is preferred? It is determined by a task. Such a medium is most often chosen, which has the model density near to the equilibrium state. If this is not obtained, then the probability of the rest particles presence or (and) the probability creating of the rest particles need to be changed. For our example, methane corresponds to the HPP2rp medium with the following parameters. The probabilities of the rest particles and probabilities the moving ones presence the initial particle distribution are $p_1^{mov} = p_2^{mov} = p_3^{mov} = 1$, $p_4^{mov} = 1$, $p_1^{rp} = 0,5$, $p_2^{rp} = 0,44$. The collision rules are equiprobable.

3.6 The Simulation of 1D Sound Wave Propagation Process in Inhomogeneous Media

In experiments, inhomogeneous medium is formed from two gases: helium (light medium) and methane (heavy medium). Here the light medium is the HPP media in at equilibrium state. The heavy medium is the HPP2rp medium with the above-listen parameters. Demarcation line of between two media runs the length of the 220-th collum of the array. As opposed to an explicit finite-difference

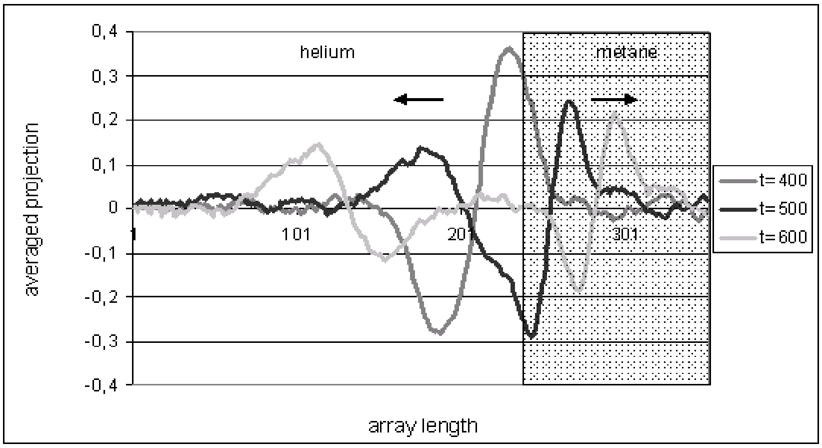


Fig. 10. There twice averaged projection of velocity vector onto the axis OX at the demarcation line

method of the sound wave simulation in inhomogeneous media, the LGA method does not require additional boundary conditions and collision rules on the boundary of discontinuity of density. Figure 10 shows he dependence of twice projection of the velocity vector onto the axis X the averaged flow change in helium-methane medium. As soon as the sound wave reaches the demarcation line, two effects (a refraction and a reflection) are observed. An experiment demonstrated that the model velocities of the refracted and reflected sound waves are equal in magnitude to the sound wave model velocities in the media corresponding to helium and to methane, respectively.

4 Conclusion

In this paper, the model wave velocity dependence of the HPPrp model parameters (number of the rest particles, density of the moving particles, the probability of the rest particles presence, and probabilities of creating and annihilating of the rest particles) are experimentally investigated. Technique for evaluation the HPPrp medium and its parameters by physical velocity of the sound wave propagation in the given medium and vice versa is given. The sound wave propagation process in inhomogeneous media is formed from two gases (helium and methane) is simulated. Experiments showed the following.

– The LGA method does not require additional conditions on the boundary of discontinuity of density for sound wave simulation in inhomogeneous media, as opposed to an explicit finite-difference method of wave equation solution.
– The sound wave velocity in the given HPPrp medium and velocity in the same medium after the boundary of discontinuity coincide very closely.

References

1. Hardy, J., Pomean, Y., de Pazzis, O.: Time evolution of a two-dimensional model system. Journal of Math. Physics 14, 1746–1752 (1973)
2. Zhang, M., Cule, D., Shafai, L., Bridges, G., Simons, N.: Computing electromagnetic fields in inhomogeneous media using lattice gas automata. In: Proceedings of 1998 Symposium on Antenna Technology and Applied Electromagnetics, Ottawa, Canada, August 14–16 (1998)
3. Simons, N.R.S., Bridges, G., Cuhachi, M.: A Lattice-Gas Automaton Capable of Modeling Three-Dimentional Electromagnetic Fields. Journal of Computational Physics 151, 816–835 (1999)
4. Markova, V.: Sound wave propagation simulation in an ingomogeneous medium using Lattice Gas Automata. Bull. Novosibirsk Comp. Center. Ser. Computer Science (27), 71–81 (2008)

Cellular-Automaton Simulation of a Cumulative Jet Formation

Yu. Medvedev

Institute of Computational Mathematics and Mathematical Geophysics,
Supercomputer Software Dept., Academican Lavrentiev ave. 6,
630090, Novosibirsk, Russia
medvedev@ssd.sscc.ru

Abstract. A new cellular-automaton FHP-GWC model is proposed. Computing experiments have been carried out with this model; they demonstrate a correlation of the new model with the physical laws.

Keywords: Cellular automata, gas flow, cumulative jet.

1 Introduction

One of perspective directions of physical processes simulation is using cellular automata (CA). CA models of flows called the Lattice-Gas have been suggested in the seventies the last century [1] and since then are promptly advanced. These models are discrete; their ground is the Boolean algebra. They allow to construct efficient programs and to minimize computer time usage.

The FHP (Frish, Hasslacher, Pomeau) model introduced in [1] is a Lattice-Gas CA is used for simulating viscous fluid flow. The FHP-MP (multiparticle FHP) model [2] allows simulating fluids with less viscosity than the FHP model because of using wider range of pressure.

The paper aims at expanding a Lattice-Gas flow model 1) to simulate a gas flow carrying several powdered components, and 2) usage of initial conditions with high pressure gas for having possibility to achieve a high velocity of the flow like by an explosion. These two innovations have given the chance to simulate the cumulative jet.

In the paper, the new model FHP-GWC (FHP with multiparticle gas and fine powders of W, C, and WC) is proposed, and results of its experimental research are described. An example of a formation of a cumulative jet with a tungsten powder is given.

2 Model Specification

The FHP-GWC model is a CA, which is represented as a triplet $\langle \mathrm{K}, N, \Theta \rangle$, where K is a set of the *cells*, N is the set of the *neighbors*, Θ is a set of the *transition functions*.

The set $\mathrm{K} = \{c_1, c_2, \ldots, c_i, \ldots\}$ consists of the *cells* allocated in corresponding sites in some discrete space. Each cell $c \in \mathrm{K}$ is described by a state $s(c)$ and two coordinates

$x(c)$ and $y(c)$ on the Cartesian plane. Therefore, between any two cells $c_1 \in$ K and $c_2 \in$ K it is easy to calculate the distance $d(c_1, c_2)$. State $s(c)$ of a cell $c \in$ K depends on a discrete clock t. Coordinates $x(c)$ and $y(c)$ of a cell $c \in$ K are time-independent. At the FHP-GWC model the state $s(c)$ is a set of one integer vector and three Boolean ones. That is because this model is the quite new model different from the FHP and the FHP-MP models.

For each cell $c \in$ K an ordered set $N(c) = \{n_i(c)$: $n_0(c) = c$, $n_i(c) \in$ K & $d(c, n_i(c)) = 1$, $(i = 1, 2, \ldots, b)\}$ is determined. Its terms belong to *a neighborhood* with the cell c and they are called its *neighboring cells* or *neighbors*. The constant b characterizes the number of non-identical neighbors of each cell $c \in$ K. Each cell is a neighbor to itself, i.e. $n_0(c) = c$. Thus, the number of neighbors of each cell c of the FHP-GWC model is equal to seven. There is a correspondence between the outputs in a cell $c \in$ K and the inputs of neighbors of this cell and vice versa. Thus, a structure of the CA cells set K is a graph in which vertices are cells, and edges from the set is the neighborhood relation. This graph has a regular lattice and degree of its vertices is equal to b.

In the FHP-GWC model, the CA with a synchronous operation is used. In each cycle (iteration), there is a replacement of states $s(t)$ in all cells $c \in$ K, by the states $s(t+1) = \theta(s(t))$, where $\theta(s(t)) \in \Theta$ is suitable next-state function. So, Θ is the set of transition functions, which define the CA evolution.

A state s of a cell $c \in$ K is a set $s(c) = \{s^G(c), s^W(c), s^C(c), s^{WC}(c)\}$ consisted of three vectors. The first $s^G(c)$ has integer components $s_i^G(c)$, $i = 0, 1, \ldots, b$, determining a number of gas particles in the cell c with *unit velocity vector* $\vec{e}_i(c)$, directed towards the neighbor $n_i(c)$ (for $i = 1, \ldots, b$) or equal to zero (for $i = 0$). Vectors $s^W(c) = (s_0^W(c), s_1^W(c), \ldots, s_b^W(c))$, $s^C(c) = (s_0^C(c), s_1^C(c), \ldots, s_b^C(c))$, and $s^{WC}(c) = (s_0^{WC}(c), s_1^{WC}(c), \ldots, s_b^{WC}(c))$ are Boolean; their components determine presence or absence of tungsten, carbon, and carbide particles with velocity vectors $M_W\vec{e}_i(c)$, $M_C\vec{e}_i(c)$, and $M_{WC}\vec{e}_i(c)$ respectively, where $i = 0, 1, \ldots, b$; M_W, M_C, and M_{WC} are specific weights of tungsten, carbon, and carbide particles. No more than one of each velocity vector $s_i^W(c)\vec{e}_i(c)$, $s_i^C(c)\vec{e}_i(c)$, and $s_i^{WC}(c)\vec{e}_i(c)$ can be directed to the neighbor $n_i(c)$. A set of states $s(c)$ of all cells $c \in$ K in the same instant t is called *a global state* $\sigma(t) = \{s(c_1), s(c_2), \ldots, s(c_i), \ldots\}$ of the CA.

A cell c, the unit velocity vectors $\vec{e}_i(c)$, and the set of the neighbors $n_i(c)$, $i = 0, 1, \ldots, 6$ are given in Fig. 1. The total particles mass moving in the direction $\vec{e}_i(c)$ is equal to $m_i(c) = s_i^G(c) + M_W s_i^W(c) + M_C s_i^C(c) + M_{WC} s_i^{WC}(c)$.

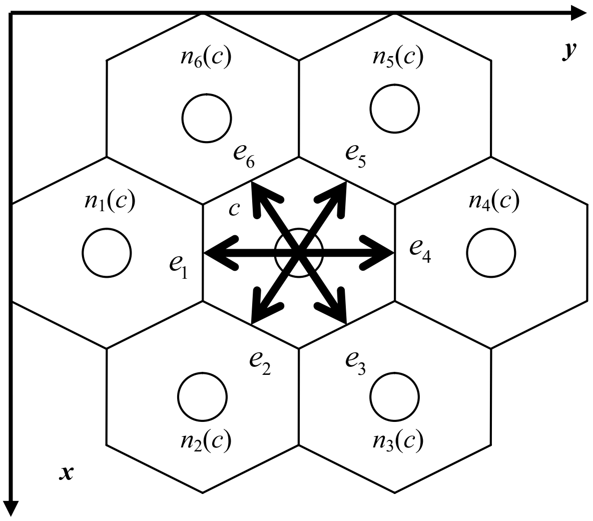


Fig. 1. The neighborhood and the unit vectors

The mass of gas particles in a cell c is equal to:

$$m^G(c) = \sum_{i=0}^{b} s_i^G(c), \tag{1}$$

where $b = 6$ is a number of possible directions of velocity vector, s_i^G is the ith component of the gas states vector s^G. A physical interpretation of the vector $s^G(c)$ components values is the following: s_i^G defines the number of unit mass particles of the gas, whose velocity vector $s_i^G \vec{e}_i$ is directed towards the neighbor $n_i(c)$. The tungsten particle mass is in M_W times larger than the gas particle mass, so:

$$m^W(c) = M_W \sum_{i=0}^{b} s_i^W(c). \tag{2}$$

The carbon particle mass is in M_C times larger than the gas particle mass, so:

$$m^C(c) = M_C \sum_{i=0}^{b} s_i^C(c). \tag{3}$$

The carbide particle mass is in M_{WC} times larger than the gas particle mass, so:

$$m^{WC}(c) = M_{WC} \sum_{i=0}^{b} s_i^{WC}(c). \tag{4}$$

Thus, the total mass of all particles in the cell c is equal to:

$$m(c) = m^G(c) + m^W(c) + m^C(c) + m^{WC}(c). \tag{5}$$

The model momentum $\vec{p}(c)$ in a cell $c \in$ K is the total momentum of all particles $\vec{p}_i(c) = \vec{p}_i^G(c) + \vec{p}_i^W(c) + \vec{p}_i^C(c) + \vec{p}_i^{WC}(c)$, including gas particles $\vec{p}_i^G(c) = s_i^G(c)\vec{e}_i(c)$, tungsten particles $\vec{p}_i^W(c) = M_W s_i^W(c)\vec{e}_i(c)$, carbon particles $\vec{p}_i^C(c) = M_C s_i^C(c)\vec{e}_i(c)$, and carbide particles $\vec{p}_i^{WC}(c) = M_{WC} s_i^{WC}(c)\vec{e}_i(c)$, directed to all neighbors $n_i(c)$, where $i = 0, 1, \ldots, b$, and $b = 6$:

$$\vec{p} = \sum_{i=1}^{b} \vec{p}_i, \tag{6}$$

From (1), with allowance for Fig. 1, it is easy to compute the total momentum $\vec{p}$ of the projections p_x and p_y onto Cartesian axes Ox and Oy:

$$p_x = \frac{\sqrt{3}}{2}\left(|\vec{p}_2| + |\vec{p}_3| - |\vec{p}_5| - |\vec{p}_6|\right), \tag{7}$$

$$p_y = |\vec{p}_4| - |\vec{p}_1| + \frac{1}{2}\left(|\vec{p}_3| + |\vec{p}_5| - |\vec{p}_2| - |\vec{p}_6|\right). \tag{8}$$

We introduce three types of cells $c \in$ K. As *conventional* cells $c_c \in \mathrm{K}_c$ we call those ones, in which both mass and momentum conservation laws are satisfied. *The wall* cells $c_w \in \mathrm{K}_w$ (walls) are the cells, in which mass conservation law is satisfied, but momentum conservation law can be violated. And, finally, *source cells* $c_s \in \mathrm{K}_s$ (sources) are cells, in which both the law of mass conservation and the law of momentum conservation can be violated. Sets of conventional cells K_c, of walls K_w, and of sources K_s do not pairwise intersect ($\mathrm{K}_c \cap \mathrm{K}_w = \varnothing$, $\mathrm{K}_c \cap \mathrm{K}_s = \varnothing$, $\mathrm{K}_w \cap \mathrm{K}_s = \varnothing$). Integration of these sets coincides with set of all cells of the CA ($\mathrm{K}_c \cup \mathrm{K}_w \cup \mathrm{K}_s = \mathrm{K}$). The behavior of walls and sources specifies the boundary conditions of the CA.

Each iteration of the CA evolution has two phases: *propagation* and *collision*. So, next-state function θ of a cell $c \in$ K consists of a superposition of the propagation function θ_1 and the collision function θ_2:

$$\theta(s(c)) = \theta_2(\theta_1(s(c))). \tag{9}$$

Both functions θ_1 and θ_2 should satisfy the laws of mass and momentum conservation:

$$\sum_{c\in \mathrm{K}}\sum_{i=0}^{b}\theta_j\left(s_i^{\mathrm{M}}(c)\right)=\sum_{c\in \mathrm{K}}\sum_{i=0}^{b}s_i^{\mathrm{M}}(c), \tag{10}$$

$$\sum_{c\in \mathrm{K}}\sum_{i=1}^{b}\theta_j\left(\vec{p}_i(c)\right)=\sum_{c\in \mathrm{K}}\sum_{i=1}^{b}\vec{p}_i(c), \tag{11}$$

where $\mathrm{M}\in\{G,W,C,WC\}$ is a type of a substance (medium), and $j\in\{1,2\}$ is the type of function.

In the propagation phase, in each cell $c \in \mathrm{K}$ each particle specified by components $s_i^{\mathrm{M}}(c)$ of state vectors $\vec{s}^{\,\mathrm{M}}(c)$, at i = 1, …, 6, propagates to the neighboring cell $n_i(c)$ corresponding to its velocity vector $\vec{e}_i(c)$. The rest particles corresponding to $s_0^{\mathrm{M}}(c)$, remain in the cell c. Thus, the i-th component $s_i^{\mathrm{M}}(c)$ of state vectors $\vec{s}^{\,\mathrm{M}}(c)$ of the cell c after propagation adopts a value:

$$\theta_1\left(s_i^{\mathrm{M}}(c)\right)=\begin{cases} s_i^{\mathrm{M}}\left(N_{((i+2)\bmod 6)+1}(c)\right), & \text{if } i=1,2,\ldots,b;\\ s_i^{\mathrm{M}}(c), & \text{if } i=0.\end{cases} \tag{12}$$

In spite of the fact that at propagation phase mass and momentum of particles in a single cell are changed, within the whole CA they are maintained, i.e. requirements (10) and (11) are fulfilled.

In the collision phase, there is a veering of particles velocity vectors directions according to some collision rules which are independent of states of the neighboring cells, i.e. θ_2 depends only on the its own state. In the FHP-GWC model the function θ_2 is probabilistic. The collision rules for the above types of the cells (conventional cells, walls, and sources) are described below.

In the conventional cells $c_c \in \mathrm{K}_c$, the function θ_2 is implemented as follows. In the beginning synthesis of the tungsten carbide is performed. For this purpose, each pair of a tungsten and a carbon particles with antiparallel velocity vectors should be converted to a particle of the tungsten carbide provided that in the direction of the tungsten velocity there is no carbide particle and there are $M_W - M_C$ or more gas particles. After synthesis, value of the function θ_2 is selected from the set of states which conserve the mass of each substance $m^{\mathrm{M}}(c_c)$ and the total momentum $\vec{p}(c_c)$:

$$\sum_{i=0}^{b}\theta_2\left(s_i^{\mathrm{M}}(c_c)\right)=\sum_{i=0}^{b}s_i^{\mathrm{M}}(c_c)-m_r^{\mathrm{M}}(c_c), \quad \forall c_c\in \mathrm{K}_c,\ \forall \mathrm{M}\in\{G,W,C,WC\}, \tag{13}$$

$$\sum_{i=1}^{b}\theta_2\left(\vec{p}_i(c_c)\right)=\sum_{i=1}^{b}\vec{p}_i(c_c),\ \forall c_c\in \mathrm{K}_c, \tag{14}$$

where $m_r^{\mathrm{M}}\left(c_c\right) \in \left\{0, M_W, M_C, -M_{WC}\right\}$ is the reaction mass in the cell c_c for various M: gas, tungsten, carbon and carbide respectively. One of the possible value obeying (13) and (14) should be chosen with equal probability. Fulfillment of (13) and (14) provides that of (10) and (11).

In the cells $c_w \in$ K$_w$, which are walls, particles are "mirrored" backwards, thus violating the momentum conservation law:

$$\theta_2\left(s_i^{\mathrm{M}}\left(c_w\right)\right) = \begin{cases} s_{((i+2)\bmod 6)+1}^{\mathrm{M}}\left(c_w\right), & \text{if } i = 1, 2, \ldots, b; \\ s_i^{\mathrm{M}}\left(c_w\right), & \text{if } i = 0. \end{cases} \tag{15}$$

Since the number of particles of every substance in a cell is not changed, requirements (13) and, therefore, (10) are satisfied. It is not so for (11), because directions of the velocity vectors $s_i^{\mathrm{M}}\left(c_w\right)\vec{e}_i\left(c_w\right)$ of the particles are changed; it is admitted by boundary conditions. Such a behavior of particles in wall cells simulates a requirement of zero speed of the flow on borders of obstacles.

Each cell-source $c_s \in$ K$_s$ sustains the given density of the gas particles $\rho_0^G\left(c_s\right)$. For this purpose, it generates particles with any possible velocity direction in case that the current density of particles $\rho^G\left(c_s\right) < \rho_0^G\left(c_s\right)$. The number of generated particles is equal to the difference between a given and a current densities $\rho_0^G\left(c_s\right) - \rho^G\left(c_s\right)$. It is possible to construct various structures of sources. For example, having placed them in one line, we can obtain a source of a steady particle flow with a given density. A single source cell simulates an injector. Naturally, when generating new particles neither the mass $m^G\left(c_s\right)$ nor the momentum $\vec{p}^G\left(c_s\right)$ are conserved. The boundary conditions in sources enable a breach of conditions (10) and (11).

In simulation, the *averaged values* of a flow velocity $\langle\vec{u}\rangle$, of the gas density $\langle\rho^G\rangle$, of the tungsten density $\langle\rho^W\rangle$, of the carbon density $\langle\rho^C\rangle$, and of the carbide density $\langle\rho^{WC}\rangle$ over some *averaging vicinity* $Av(c_0)$ in which one includes all cells $c \in$ K placed not farther from a cell c_0, than on some distance r called *the averaging radius*, have a practical significance. Also the amount of the synthesized tungsten carbide is of interest.

An averaged flow velocity is the sum of velocity vectors of all particles in the averaging vicinity $Av(c_0)$, divided by the cardinal number of the $Av(c_0)$:

$$\langle\vec{u}\rangle\left(c_0\right) = \frac{1}{\left|Av\left(c_0\right)\right|} \sum_{c \in Av\left(c_0\right)} \frac{\sum_{i=0}^{b} \vec{p}_i\left(c\right)}{\sum_{i=0}^{b} m_i\left(c\right)}, \tag{16}$$

where $|Av(c_0)|$ is the number of the cells situated in the $Av(c_0)$.

An averaged density of particles $\left\langle \rho^{\mathrm{M}} \right\rangle$ is evaluated in the same vicinity $Av(c_0)$ as follows:

$$\left\langle \rho^{\mathrm{M}} \right\rangle (c_0) = \frac{1}{\left| Av(c_0) \right|} \sum_{c \in Av(c_0)} \sum_{i=0}^{b} s_i^{\mathrm{M}}(c). \tag{17}$$

An averaged value of velocity $\left\langle \vec{u} \right\rangle$ is the model velocity of a flow. Averaged densities of the tungsten $\left\langle \rho^{W} \right\rangle$, the carbon $\left\langle \rho^{C} \right\rangle$, and the carbide $\left\langle \rho^{WC} \right\rangle$ particles are the model densities this substances. An averaged density of the gas particles $\left\langle \rho^{G} \right\rangle$ is the model pressure.

We will notice that average values of the model velocity, the model density, and the model pressure match their physical analogs only in case when the averaging vicinity $Av(c)$ consists exclusively of conventional cells $c_c \in \mathrm{K}_c$. Otherwise, we suggest the values $\left\langle \vec{u} \right\rangle$ and $\left\langle \rho^{\mathrm{M}} \right\rangle$ as indefinite. This requirement does not allow calculating the values $\left\langle \vec{u} \right\rangle$ and $\left\langle \rho^{\mathrm{M}} \right\rangle$ for the cells, which are closer to walls $c_w \in \mathrm{K}_w$ and sources $c_s \in \mathrm{K}_s$, than averaging radius r.

3 Computer Simulation

For experimental studying of the proposed model its program implementation has been performed. This allows carrying out computation experiments both on single-processor computers, and on multiprocessor or multicomputer systems. The code has been written in C, parallelism is implemented by means of the MPI library. The computation experiments with the FHP-GWC model have been performed. The qualitative behavior of a simulated cumulative jet is obtained.

The CA used in this computation experiment has the size 200 by 400 cells (along Cartesian axes Ox and Oy respectively). Its initial state is given in the Fig. 2 at the left top part. The perimeter cells with coordinates in the intervals [(1, 1), (1, 200)], [(1, 1), (400, 1)], [(1, 200), (400, 200)], and [(400, 1), (400, 200)] are walls (thin lines in the Fig. 2). The cells forming the nozzle with coordinates in the intervals [(100, 1), (100, 80)], [(100, 121), (100, 200)] are walls also. Remaining cells are conventional ones. They contain the gas particles with density $\left\langle \rho_i^{G} \right\rangle = 3$ particles per each i-th direction ($i = 0, 1, \ldots, 6$). The tungsten powder with density $\left\langle \rho_0^{W} \right\rangle = 1$ rest particles per cell is the set of cells on the left by two sloping strips with thickness equal to 10 cells. The mass of the tungsten particle $M_W = 20$. For representing explode, the tungsten strips are bordered from the outside by two high-pressure gas strips with thickness equal to 20 cells and with density $\left\langle \rho_0^{G} \right\rangle = 60$ rest particles per cell (invisible in the Fig. 2).

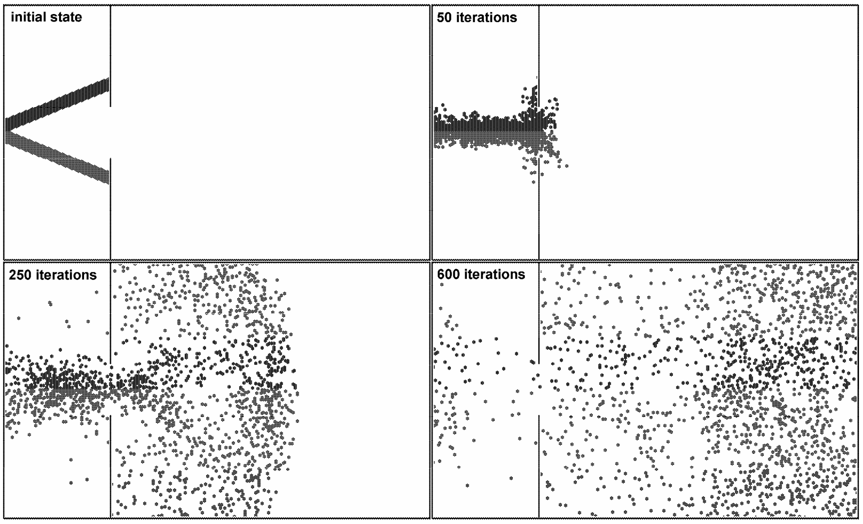


Fig. 2. Cumulative jet with a tungsten fine powder

After 50, 250 and 600 iterations the average density of tungsten $\left\langle \rho^{W} \right\rangle$ with averaging radius $r = 1$ has been obtained (Fig. 2). The jet of the tungsten powder looks similar to the obtained in natural experiments. Results of this experiment demonstrate that the FHP-GWC model correctly reproduces process in the cumulative jets and corresponds to physics.

4 Conclusion

In the paper the new cellular-automaton FHP-GWC model is proposed. The model intended for simulating cumulative powder-gas jet. The computing experiments have been carried out; they demonstrate a correlation of the new model with the physical laws.

Future work will deal with cumulative jet in which a chemical reaction of tungsten carbide synthesis in colliding two cumulative jets. Also, the parallel program realization of the model will be studied by computing on clusters.

So, simulating gas flow with powder components is requested by the investigations in the field of new composite materials.

References

1. Frisch, U., Hasslacher, B., Pomeau, Y.: Lattice-Gas Automata for Navier-Stokes Equations. Phys. Rev. Lett. N 56, 1505 (1986)
2. Medvedev, Yu.: The FHP-MP model as multiparticle Lattice-Gas. Bull. Nov. Comp. Center, Comp. Science 27, 83–91 (2008)

Associative Version of the Ramalingam Decremental Algorithm for Dynamic Updating the Single-Sink Shortest-Paths Subgraph

Anna Nepomniaschaya

Institute of Computational Mathematics and Mathematical Geophysics,
Siberian Division of Russian Academy of Sciences,
pr. Lavrentieva, 6, Novosibirsk, 630090, Russia
anep@ssd.sscc.ru

Abstract. We propose an efficient implementation of the Ramalingam algorithm for dynamic updating the single-sink shortest-paths subgraph of a directed weighted graph after deletion of an edge using a model of associative (content addressable) parallel systems with vertical processing (the STAR–machine). The associative version of the Ramalingam decremental algorithm is given as the procedure DeleteArc, whose correctness is proved and the time complexity is evaluated. We compare implementations of the Ramalingam decremental algorithm and its associative version and present the main advantages of the associative version.

Keywords: Directed weighted graph, subgraph of the shortest paths, adjacency matrix, decremental algorithm, associative parallel processor, access data by contents.

1 Introduction

Finding the shortest paths in a weighted graph is a fundamental and well studied problem in computer science. Such a problem arises in practice in different application settings. There are two versions of this problem: finding the single source shortest paths and finding the all–pairs shortest paths.

The dynamic version of the single source shortest paths problem consists of maintaining the shortest paths information while the graph changes without recomputing everything from scratch after every update on the graph. The most general types of update operations for the single source shortest paths problem include insertions and deletions of edges and update operations on the edge weights. When arbitrary sequences of the above operations are allowed, we refer to the *fully dynamic* problem. If we consider only insertions (deletions) of edges, we refer to the *incremental* (*decremental*) problem.

In the case of arbitrary real edge weights, Ramalingam and Reps [10, 11] devise fully dynamic algorithms for updating the single source shortest paths using the output bounded model. In this model, the running time of an algorithm is analyzed in terms of the output change rather than the input size. The authors assume that the graph has no negative–length cycles before and after input update. Frigioni et al. [4] study the semi–dynamic single source shortest paths problem for both directed and undirected graphs with positive real edge weights in terms of the output complexity. Frigioni et al. [3] propose fully dynamic algorithms for updating the distances and a single source shortest paths tree (*sp-tree*) in either a directed or an undirected graph with positive real edge weights under arbitrary sequences of edge updates. The cost of the update operations is given as a function of the number of output updates by using the notion of k-bounded accounting function. Frigioni et al. [5] propose the fully dynamic solution for the problem of updating the shortest paths from a given source in a directed graph with arbitrary edge weights. The authors devise a new algorithm for performing edge deletions and weight increases that explicitly deals with zero–length cycles. Algorithms from [3–5, 10, 11] use the dynamic version of the Dijkstra algorithm [1]. Narváez et al. [6] propose two incremental methods to transform the well–known static algorithms of Dijkstra, Bellman-Ford, and D'Esopo-Pape into new dynamic algorithms for updating an sp-tree after changing edge weights.

In this paper, we deal with a directed graph G and the shortest-paths subgraph $SP(G)$ that consists of all shortest paths from every vertex to the sink. We propose an associative version of the Ramalingam algorithm [10] for the dynamic update of $SP(G)$ after deletion of an edge from G. Our model of computation (the STAR–machine) simulates the run of associative (content addressable) parallel systems of the SIMD type with bit–serial (vertical) processing. Such an architecture is best suited to solve the graph problems. We first offer a simple and natural data structure for efficient implementation of the Ramalingam decremental algorithm on the STAR–machine. The associative version of this algorithm is given as the procedure DeleteArc, whose correctness is proved. We obtain that this procedure takes $O(hk)$ time, where h is the number of bits for coding the infinity and k is the number of vertices, whose shortest paths to the sink change after deleting an edge from $SP(G)$. Following [2], we assume that each elementary operation of the STAR–machine (its microstep) takes one unit of time. We also present the main advantages of the associative version of the Ramalingam decremental algorithm.

2 Model of Associative Parallel Machine

Here, we propose a brief description of our model. It is defined as an abstract STAR–machine of the SIMD type with the vertical data processing [7]. It consists of the following components:

– a sequential control unit (CU), where programs and scalar constants are stored;

– an associative processing unit consisting of p single–bit processing elements (PEs);

– a matrix memory for the associative processing unit.

The CU passes an instruction to all PEs in one unit of time. All active PEs execute it in parallel, while inactive PEs do not perform it. Activation of a PE depends on data.

Input binary data are given in the form of two–dimensional tables, where each datum occupies an individual row and is updated by a dedicated PE. In any table, rows are numbered from top to bottom and columns – from left to right. Some tables may be loaded into the memory.

An associative processing unit is represented as h vertical registers, each consisting of p bits. Vertical registers can be regarded as a one-column array. The bit columns of the tabular data are stored in the registers that perform the necessary Boolean operations.

The run is described by means of the language STAR being an extension of Pascal. Let us briefly consider the STAR constructions needed for the paper. To simulate the data processing in the matrix memory, we use the data types **word, slice**, and **table**. Constants for the types **slice** and **word** are represented as a sequence of symbols of a set $\{0, 1\}$ enclosed within single quotation marks. The types **slice** and **word** are used for the bit column access and the bit row access, respectively, and the type **table** is used for defining the tabular data. Assume that any variable of the type **slice** consists of p components, which belong to $\{0, 1\}$. For simplicity, let us call *slice* any variable of the type **slice**.

Let us present some elementary operations and a predicate for slices.

Let X, Y be variables of the type **slice** and i be a variable of the type **integer**. We use the following operations:

SET(Y) simultaneously sets all components of Y to $'1'$;

CLR(Y) simultaneously sets all components of Y to $'0'$;

$Y(i)$ selects the i-th component of Y;

FND(Y) returns the number i of the first (the uppermost) $'1'$ of Y, $i \geq 0$;

STEP(Y) returns the same result as FND(Y), then resets the first $'1'$ found to $'0'$;

CONVERT(Y) returns a row, whose every i-th bit coincides with $Y(i)$. It is applied when a row of one matrix is used as a slice for another matrix.

The operations FND(Y), STEP(Y), and CONVERT(Y) are used only as the right part of the assignment statement, while the operation $Y(i)$ is used as both the right part and the left part of the assignment statement.

To carry out the data parallelism, we introduce in the usual way the bitwise Boolean operations: $X\, and\, Y$, $X\, or\, Y$, $not\, Y$, $X\, xor\, Y$. We also use a predicate SOME(Y) that results in **true** if there is at least a single bit $'1'$ in the slice Y.[1]

Note that the predicate SOME(Y) and all operations for the type **slice** are also performed for the type **word**. We will also employ the bitwise Boolean operations between a variable w of the type **word** and a variable Y of the type **slice**, where the number of bits in w coincides with the number of bits in Y.

Let T be a variable of the type **table**. We employ the following operations:

ROW(i, T) returns the i-th row of the matrix T;

COL(i, T) returns its i-th column.

[1] For simplicity, the notation $Y \neq \Theta$ denotes that the predicate SOME(Y) results in **true**.

Note that the STAR statements are defined in the same manner as for Pascal.

Now, we recall a group of basic procedures [8, 9] implemented on the STAR–machine which will be used later on. These procedures use the given slice X to indicate with $'1'$ the row positions used in the corresponding procedure.

The procedure MIN(T, X, Z) defines positions of rows in the given matrix T where the minimal element is located. These positions are marked with $'1'$ in the result slice Z.

The procedure SETMIN(T, F, X, Z) defines positions of the matrix T rows that are less than the corresponding rows of the matrix F. It returns the slice Z, where $Z(i) =' 1'$ if ROW$(i, T) <$ ROW(i, F) and $X(i) =' 1'$.

The procedure WCOPY(v, X, F) writes the given binary word v into those rows of the matrix F, that correspond to positions $'1'$ in the slice X. Other rows of the matrix F consist of zeros.

The procedure TCOPY1(T, j, h, F) writes h columns from the given matrix T, starting from the $(1 + (j - 1)h)$-th column, into the result matrix F $(j \geq 1)$.

The procedure HIT(T, F, X, Z) defines positions of the corresponding identical rows in the given matrices T and F using the slice X. These positions are marked with $'1'$ in the result slice Z.

The procedure ADDV(T, F, X, R) writes into the matrix R the result of parallel addition of the corresponding rows of matrices T and F, whose positions are selected with $'1'$ in the slice X. This algorithm uses the table 5.1 from [2].

The procedure ADDC(T, X, v, F) adds the binary word v to the rows of the matrix T selected with $'1'$ in X, and writes down the result into the corresponding rows of the matrix F. Other rows of the matrix F are set to zero.

The procedure TMERGE(T, X, F) writes the rows of the matrix T, that correspond to positions $'1'$ in the slice X, into the matrix F. Other rows of the matrix F are not changed.

In [8, 9], we have shown that these procedures take $O(h)$ time each, where h is the number of bit columns in the corresponding matrix.

3 Preliminaries

Let $G = (V, E, w)$ be a *directed weighted graph* with the set of vertices $V = \{1, 2, \ldots, n\}$, the set of directed edges (arcs) $E \subseteq V \times V$ and the function w that assigns a weight to every edge. We will consider graphs with a distinguished vertex s called *sink*.

An *adjacency matrix Adj* $= [a_{ij}]$ of a directed graph G is an $n \times n$ Boolean matrix, where $a_{ij} = 1$ if and only if there is an arc from the vertex i to the vertex j in the set E.

An arc e directed from i to j is denoted by $e = (i, j)$, where the vertex i is the *head* of e (or $father$) and the vertex j is its *tail* (or *son*). We assume that all arcs have a positive weight and $w(u, v) = \infty$ if $(u, v) \notin E$. Let h be the number of bits for coding the infinity.

A *path* from u to s in G is a finite sequence of vertices $u = v_1, v_2, \ldots, v_k = s$, where $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k - 1$ and $k > 1$. The *shortest path* from u to s is the path of the minimal sum of weights of its arcs.

Let $dist(u)$ denote the length of the shortest path from u to s and $SP(G)$ denote the subgraph of the shortest paths from all vertices of G to the sink.

By analogy with Ramalingam, we introduce the following notations.

We denote by *outdegree*(v) the number of arcs outgoing from the vertex v in $SP(G)$. Let an arc (i, j) be deleted from $SP(G)$. Let $AffectedV$ denote the set of all vertices u in $SP(G)$ such that all paths from u to the sink include the deleted arc (i, j). An arc (x, y) is called $affected$ by deleting the arc (i, j) in $SP(G)$ if there is no such path from x to s in the new graph that uses the arc (x, y) and the length of the path is equal to $dist_{old}(x)$.

4 The Ramalingam Decremental Algorithm for the Single-Sink Shortest Paths Problem

Let an arc (i, j) be deleted from $SP(G)$ and *outdegree*$(i) = 0$.

The Ramalingam decremental algorithm for dynamic updating the single-sink shortest-paths subgraph consists of the following two stages.

At the *first stage*, one determines the set $AffectedV$ and all affected arcs obtained after deleting the arc (i, j) from $SP(G)$. Then affected arcs are deleted from $SP(G)$.

At the *second stage*, for every affected vertex v_i, one computes a new shortest path from v_i to s in G and updates $SP(G)$.

The *first stage* is performed as follows.

Initially, $AffectedV = \Theta$. To construct it, an auxiliary set of vertices $WorkSet$ is used. Initially, $WorkSet = \{i\}$ because *outdegree*$(i) = 0$ after deleting the arc (i, j) from $SP(G)$. Vertices in $WorkSet$ are sequentially updated. The current updated vertex u is deleted from $WorkSet$ and is included into the set $AffectedV$. Then every arc (x, u) entering the vertex u is deleted from $SP(G)$ and *outdegree*(x) is decreased by one. If *outdegree*$(x) = 0$, the vertex x is included into $WorkSet$.

To perform the *second stage*, one uses a heap *PriorityQueue*, whose elements are affected vertices with a key. At this stage, one first computes for every affected vertex u such a new shortest path to the sink that does not include other affected vertices. The value of $dist(u)$ is its current key in the heap. After that one updates $SP(G)$ as follows.

At every iteration, a vertex with the minimum key in the heap (say a) is deleted from the set *PriorityQueue*. Then one determines those arcs (a, b) that belong to an alternative path from the vertex a to the sink and $dist_{new}(a) = w(a, b) + dist_{old}(b)$. Every such arc (a, b) is included into $SP(G)$ and *outdegree*(a) is increased by one. Further all arcs (c, a) are analyzed. If a new path from the vertex c to the sink includes the arc (c, a) and $dist_{new}(c) < dist_{old}(c)$, the current value $dist(c)$ is equal to $dist_{new}(c)$ and this value is the new key for the vertex c in *PriorityQueue*.

The process is completed after updating all vertices in the heap.

5 Associative Version of the Ramalingam Decremental Algorithm

To design an associative version of the Ramalingam decremental algorithm, we employ the following data structure:

– an $n \times n$ adjacency matrix G, whose every i-th column saves with $'1'$ the tails of arcs outgoing from the vertex i;

– an $n \times n$ adjacency matrix SP, whose every i-th column saves with $'1'$ the tails of arcs outgoing from the vertex i that belong to the shortest-paths subgraph;

– an $n \times hn$ matrix $Weight$ that contains as elements the arc weights. It consists of n fields having h bits each. The weight of an arc (i, j) is written in the j-th row of the i-th field;

– an $n \times hn$ matrix $Cost$ that contains as elements the arc weights. It consists of n fields having h bits each. The weight of an arc (i, j) is written in the i-th row of the j-th field;

– an $n \times h$ matrix $Dist$, whose every i-th row saves the shortest distance from the vertex i to the sink;

– a slice $AffectedV$ that saves with $'1'$ positions of all affected vertices.

Note that the i-th field of the matrix $Weight$ saves the weights of arcs *outgoing* from the vertex i, while the i-th field of the matrix $Cost$ saves the weights of arcs *entering* the vertex i.

We will use the following property of the matrices G and SP.

Property 1. In every i-th row of the matrices G and SP, the heads of arcs entering the vertex i are marked with $'1'$.

Let an arc (i, j) be deleted from G and SP.

We first provide an associative parallel algorithm (say *Algorithm A*) for selecting the set of affected vertices and arcs. This algorithm uses the slices WS and $AffectedV$ and the matrix SP. It performs the following steps.

Step 1. Set zeros into the slices $AffectedV$ and WS. Check whether there is an arc outgoing from the vertex i in SP. If it is true, go to exit. Otherwise, include the vertex i into WS.

Step 2. While $WS \neq \Theta$, perform the following actions:

– delete the position of the first $'1'$ (say k) from the slice WS. Include the vertex k into the slice $AffectedV$;

– delete all arcs from SP that enter the vertex k;

– for every deleted arc (r, k), include the vertex r into the slice WS if there is no arc outgoing from r in SP.

On the STAR–machine, this algorithm is implemented as the procedure FindAffectedVert.

An associative parallel algorithm for finding a new distance to the sink from an affected vertex k (say *Algorithm B*) uses the slice $AffectedV$ and the matrices G and $Dist$. It runs as follows.

Step 1. Compute *in parallel* distances from the vertex k to s for every path in the matrix G that begins with an arc (k, r), where $r \notin AffectedV$.
Step 2. Select the minimum distance from k to s and write it down into the k-th row of the matrix $Dist$.

On the STAR–machine, this algorithm is implemented as the procedure ComputeNewDist.

An associative parallel algorithm for updating arcs outgoing from an affected vertex k (say *Algorithm C*) uses the slices Z and Y, and the matrices G, SP, and $Dist$. It performs the following steps.

Step 1. By means of a slice (say Z), save the positions of all arcs outgoing from the vertex k in the matrix G.
Step 2. Determine *in parallel* distances from the vertex k to the sink for different paths that include an arc marked with $'1'$ in the slice Z.
Step 3. By means of a slice (say Y), save positions of those arcs (k, l) for which $dist(k) = w(k, l) + dist(l)$.
Step 4. Include positions of arcs marked with $'1'$ in the slice Y into SP.

On the STAR–machine, this algorithm is implemented as the procedure UpdateOutgoingArcs.

An associative parallel algorithm for updating arcs entering an affected vertex k (say *Algorithm D*) uses the slices Z and Y and the matrices G and $Dist$. It performs the following steps.

Step 1. By means of a slice (say Z), save the *heads* of arcs entering the vertex k in G.
Step 2. For all vertices l marked with $'1'$ in the slice Z, determine *in parallel* distances to the sink in every path starting with the arc (l, k).
Step 3. By means of a slice (say Y), save *positions* of those vertices r, marked with $'1'$ in the slice Z, for which $dist_{new}(r) < dist_{old}(r)$. Then write $dist_{new}(r)$ in the corresponding rows of the matrix $Dist$.

On the STAR–machine, this algorithm is implemented as the procedure UpdateIncomingArcs.

Now, we provide an associative parallel algorithm for updating the shortest-paths subgraph after deletion of the arc (i, j) from the matrix G. It performs the following steps.

Step 1. Delete the *position* of the arc (i, j) from the matrix G. If $(i, j) \notin SP$, go to exit. Otherwise, delete the *position* of this arc from the matrix SP.
Step 2. By means of the *Algorithm A*, construct the slice $AffectedV$ and delete affected arcs from SP. Save a copy of the slice $AffectedV$ in another slice (say X).

Step 3. While $X \neq \Theta$, determine new distances to the sink from all affected vertices as follows:

– select the position of the current affected vertex k in the slice X and mark it with $'0'$;

– by means of the *Algorithm B*, determine the new distance from the vertex k to the sink.

Step 4. While $AffectedV \neq \Theta$, update affected vertices taking into account their new distances to the sink as follows:

– knowing the slice $AffectedV$ and the matrix $Dist$, determine the position of an affected vertex q having the minimum distance to the sink and mark this position with $'0'$ in $AffectedV$;

– by means of the *Algorithm C*, update the arcs outgoing from the vertex q;

– by means of the *Algorithm D*, update the arcs entering the vertex q.

On the STAR–machine, this algorithm is given as the procedure DeleteArc.

6 Implementation of the Associative Version of the Ramalingam Decremental Algorithm

In this section, we first briefly explain the run of the auxiliary procedures. Then we propose the procedure DeleteArc. In a full paper, we will provide the detailed analysis of the auxiliary procedures and their correctness.

We first consider the procedure FindAffectedVert. Knowing the head i of the deleted arc (i, j) and the matrix SP, it returns the updated matrix SP and the slice $AffectedV$, where positions of all affected vertices are marked with $'1'$.

Let the current vertex k be included into the slice $AffectedV$. To delete the arcs entering the vertex k and to update their heads, we first save the k-th row of the matrix SP by means of a variable v of the type **word**. Then we write zeros in the k-th row of SP. While $v \neq \Theta$, by means of the operation STEP(v), we select the leftmost head r and check whether COL(r, SP) consists of zeros.

Now we explain the procedure ComputeNewDist. Knowing the integers h and k, the slice $AffectedV$ and the matrices G, $Weight$, and $Dist$, it returns the updated matrix $Dist$.

To determine a new distance in G from the vertex k to s, we first save the sons of k that are not affected. Then by means of TCOPY1, we select the k-th field of the matrix $Weight$. Further, by means of ADDV, we add the k-th field of the matrix $Weight$ and the matrix $Dist$ for the corresponding selected rows. Finally, by means of the basic procedure MIN and the operation FND, we determine the new distance from k to s and write it into the k-th row of the matrix $Dist$.

Now, we proceed to the procedure UpdateOutgoingArcs. Knowing the integers h and k, and the current matrices G, $Weight$, $Dist$, and SP, the procedure returns the updated matrix SP.

By analogy with the procedure ComputeNewDist, this procedure first saves different distances from k to s in a matrix $W2$. Then by means of WCOPY, the distance from k to s is written in the rows of a matrix $W1$ that correspond to positions of sons of the vertex k in G. Further, by means of HIT, we select the sons of k that belong to the alternative shortest paths from k to s. Finally, positions of these arcs are included into SP.

Finally, we consider the procedure UpdateIncomingArcs. Knowing the integers h and k and the current matrices G, $Cost$, and $Dist$, the procedure returns the updated matrix $Dist$.

Initially, by means of the operation CONVERT, we transform the k-th row of the matrix G into a slice Z. Then by means of TCOPY1, we select the k-th field of the matrix $Cost$. To determine the new distances to s from the heads of arcs entering the vertex k, we add $dist(k)$ to the rows of the k-th field of the matrix $Cost$, using the slice Z and the basic procedure ADDC. Then by means of SETMIN, we select the heads r of arcs entering k for which $dist_{new}(r) < dist_{old}(r)$. Finally, we write $dist_{new}(r)$ in the corresponding rows of the matrix $Dist$.

Let us proceed to the procedure DeleteArc. Knowing the deleted arc (i, j), the integer h and the current matrices G, $Weight$, $Cost$, $Dist$, and SP, the procedure returns the updated matrices G, SP, and $Dist$ with the use of the above auxiliary procedures.

```
procedure DeleteArc(i,j,h:integer;Weight,Cost:table;
  var G,SP:table; var Dist:table);
/* The arc $(i,j)$ is deleted from the matrices $G$ and $SP$. */
var k:integer;
  AffectedV,X,Y:slice(G);
  label 1;
1.Begin X:=COL(i,G); X(j):='0';
2.  COL(i,G):=X;
/* The arc $(i,j)$ is deleted from $G$. */
3.  X:=COL(i,SP);
4.  if X(j)='0' then goto 1;
5.  X(j):='0'; COL(i,SP):=X;
/* The arc $(i,j)$ is deleted from $SP$. */
6.  FindAffectedVert(i,SP,AffectedV);
/* This procedure returns the updated matrix $SP$
  and the slice $AffectedV$. */
7.  X:=AffectedV;
8.  while SOME(X) do
9.    begin k:=STEP(X);
10.     ComputeNewDist(h,k,AffectedV,G,Weight,Dist);
/* The new distance from the vertex $k$ to $s$ is written
  in the $k$-th row of the matrix $Dist$. */
11.   end;
12.   while SOME(AffectedV) do
13.     begin MIN(Dist,AffectedV,Y);
14.       k:=FND(Y); AffectedV(k):='0';
```

```
15.        UpdateOutgoingArcs(h,k,G,Weight,Dist,SP);
 /* We include into $SP$ those arcs $(k,r)$, for which
    $dist(k) = w(k,r) + dist(r)$. */
16.        UpdateIncomingArcs(h,k,G,Cost,Dist);
 /* We write $dist_{new}(l)$ into the $l$-th row of the matrix $Dist$
    if $dist_{new}(l) < dist_{old}(l)$ and the arc $(l,k)$ belongs
    to the path from $l$ to $s$. */
17.      end;
18. 1: End.
```

Theorem 1. *Let a directed weighted graph be given as an adjacency matrix G and a matrix Weight. Let matrices Cost, SP, and Dist and the number of bits h for coding the infinity be also given. Let an arc (i,j) be deleted from the graph. Then after performing the procedure* DeleteArc*, this arc is deleted from the matrices G and SP. Moreover, matrices SP and Dist are updated according to the algorithms A, B, C, and D.*

Proof. (Sketch). We prove this by induction in terms of the number q of affected vertices that appear after deleting the arc (i,j) from SP.

Basis is proved for $q = 1$. One can immediately check that after performing lines 1–5, the arc (i,j) is deleted from the matrices G and SP. After performing the procedure FinfAffectedVert (line 6), the slice $AffectedV$ saves the vertex i and all arcs, entering this vertex, are deleted from SP. After performing line 7, we have $X(i) =' 1'$. One can easily check that after fulfilling line 9, we have $k = i$ and $X = \Theta$ because initially the slice X is the copy of $AffectedV$. After performing the auxiliary procedure ComputeNewDist (line 10), the new distance from i to s is written in the i-th row of the matrix $Dist$. Since $X = \Theta$, we carry out line 12. After performing lines 12–14, we have $k = i$ and $AffectedV = \Theta$. Further, after performing the procedure UpdateOutgoingarcs (line 15), all arcs (i,r) for which $dist_{new}(i) = w(i,r) + dist_{old}(r)$ are included into SP.

By assumption, there is a single affected vertex in SP. It means that there is an alternative path to the sink for every vertex l, being the head of any arc (l,i) in SP. Therefore after performing the procedure UpdateIncomingArcs (line 16), the matrix $Dist$ does not change.

Hence, after performing the procedure Deletearc, the arc (i,j) is deleted from the matrices G and SP, $dist_{new}(i)$ is written into the i-th row of the matrix $Dist$, and all arcs (i,r), for which $dist_{new}(i) = w(i,r) + dist_{old}(r)$, are included into SP.

Step of induction. Let the assertion be true when no more than $q \geq 1$ affected vertices are updated in the given graph. We will prove the assertion for $q + 1$ affected vertices.

One can immediately verify that, after performing lines 1–7, the arc (i,j) is deleted from G and SP, the slice $AffectedV$ saves positions of $q+1$ affected vertices, affected arcs are deleted from SP, and the slice X is a copy of $AffectedV$. After performing line 9, the position of the first (or uppermost) affected vertex

k is determined. By analogy with the basis, after performing the procedure ComputeNewDist (line 10), the new distance from k to s is written into the k-th row of the matrix $Dist$. Now, only q affected vertices are marked with $'1'$ in the slice $AffectedV$. By the inductive assumption, after execution of the cycle `while SOME(X) do` (line 8), new distances from every affected vertex to s will be written in the corresponding rows of the matrix $Dist$.

Since $X = \Theta$, we carry out the cycle `while SOME(AffectedV) do` (line 12). After performing lines 13–14, we determine the position of the affected vertex k having the minimum new distance to s and mark it with $'0'$ in the slice $AffectedV$ After performing the procedure UpdateOutgoingarcs (line 15), we include into SP the positions of arcs (k, r), for which $dist_{new}(k) = w(k, r) + dist_{old}(r)$. Further, after performing the procedure UpdateIncomingArcs (line 16), for every affected vertex r, for which $dist_{new}(r) < dist_{old}(r)$, we write $dist_{new}(r)$ into the r-th row of the matrix $Dist$.

Now, there are only q affected vertices, whose positions are marked with $'1'$ in the slice $AffextedV$. By the inductive assumption, after updating q affected vertices, all alternative paths from every affected vertex r to the sink are included into SP and the new distance from r to s is written in the r-th row of the matrix $Dist$. Hence, the assertion is true for $q + 1$ affected vertices.

This completes the proof.

Let us evaluate the time complexity of the procedure DeleteArc. To this end, we first evaluate the time complexity of the auxiliary procedures. Let h be the number of bits for coding the infinity and k be the number of affected vertices that appear in $SP(G)$ after deleting the arc (i, j). The auxiliary procedure Find-AffectedVert takes $O(k)$ time. Other auxiliary procedures take $O(h)$ time each. In the procedure DeleteArc, the cycle `while SOME(X) do` (lines 8–11) and the cycle `while SOME(AffectedV) do` (lines 12–17) take $O(kh)$ time each. Hence, the procedure DeleteArc takes $O(kh)$ time.

Now we compare implementations of the Ramalingam decremental algorithm and its associative version:

– the Ramalingam decremental algorithm uses *a heap of vertices*, where the distance from any affected vertex r to the sink is its current key in the heap. The associative version saves the current distance from r to s in the r-th row of the matrix $Dist$;

– for every affected vertex r, the Ramalingam decremental algorithm determines in succession different distances from r to s and assignes the minimum distance among them to the current key for the vertex r in the heap. The associative version *simultaneously* determines different distances from r to s and writes the minimum distance into the r-th row of the matrix $Dist$;

– for every affected vertex r, the Ramalingam decremental algorithm determines in succession those arcs (r, l), for which $dist(r) = w(r, l) + dist(l)$. The associative version *simultaneously* determines *positions* of such arcs;

– for every affected vertex r, the Ramalingam decremental algorithm determines in succession those arcs (q, r), for which $dist_{new}(q) < dist_{old}(q)$, then assigns $dist_{new}(q)$ to the current key of q in the heap. The associative version *simultaneously* determines *positions* of such heads of arcs (q, r), then *simultaneously* writes the new distances from them to the sink into the corresponding rows of the matrix $Dist$.

7 Conclusions

We have proposed a new data structure for efficient implementation of the Ramalingam decremental algorithm on the STAR–machine having no less than n PEs. The associative version of the Ramalingam decremental algorithm is represented as the procedure DeleteArc, whose correctness is proved. We have obtained that this procedure takes $O(kh)$ time per a deletion, where h is the number of bits for coding the infinity and k is the number of affected vertices that appear in $SP(G)$ after deleting an arc. It is assumed that each microstep of the STAR–machine takes one unit of time. We have also compared the implementations of the Ramalingam decremental algorithm and its associative version and presented the main advantages of the associative version.

We are planning to design an associative version of the Ramalingam incremental algorithm for the dynamic update of the shortest-paths subgraph after insertion of an arc into the given graph.

References

1. Dijkstra, E.W.: A Note on Two Problems in Connection with Graphs. Numerische Mathematik 1, 269–271 (1959)
2. Foster, C.C.: Content Addressable Parallel Processors. Van Nostrand Reinhold Company, New York (1976)
3. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully Dynamic Algorithms for Maintaining Shortest Paths Trees. J. of Algorithms 34, 351–381 (2000)
4. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Semi–dynamic Algorithms for Maintaining Single Source Shortest Paths Trees. Algorithmica 25, 250–274 (1998)
5. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully Dynamic Shortest Paths in Digraphs with Arbitrary Arc Weights. J. of Algorithms 49, 86–113 (2003)
6. Narváez, P., Siu, K.-Y., Tzeng, H.-Y.: New Dynamic Algorithms for Shortest Paths Tree Computation. IEEE/ACM Trans. Networking. 8, 734–746 (2000)
7. Nepomniaschaya, A.S.: Language STAR for Associative and Parallel Computation with Vertical Data Processing. In: Mirenkov, N. (ed.) Proc. of the Intern. Conf. Parallel Computing Technologies, pp. 258–265. World Scientific, Singapore (1991)
8. Nepomniaschaya, A.S.: Solution of Path Problems Using Associative Parallel Processors. In: Intern. Conf. on Parallel and Distributed Systems, ICPADS 1997, pp. 610–617. IEEE Press, New York (1997)
9. Nepomniaschaya, A.S., Dvoskina, M.A.: A simple Implementation of Dijkstra's Shortest Path Algorithm on Associative Parallel Processors. Fundamenta Informaticae 43, 227–243 (2000)
10. Ramalingam, G.: Bounded Incremental Computation. LNCS, vol. 1089. Springer, Heidelberg (1996)
11. Ramalingam, G., Reps, T.: An Incremental Algorithm for a Generalization of the Shortest Paths Problem. J. of Algorithms 21, 267–305 (1996)

Cellular Automata-Based S-Boxes vs. DES S-Boxes

Miroslaw Szaban[1] and Franciszek Seredynski[2,3]

[1] Institute of Computer Science, University of Podlasie
3-go Maja 54, 08-110 Siedlce, Poland
mszaban@ap.siedlce.pl

[2] Institute of Computer Science, Polish Academy of Sciences,
Ordona 21, 01-237 Warsaw, Poland

[3] Polish-Japanese Institute of Information Technology
Koszykowa 86, 02-008 Warsaw, Poland
sered@ipipan.waw.pl

Abstract. In the paper we use recently proposed cellular automata (CA) - based methodology [9] to design 6x4 S-boxes functionally equivalent to S-boxes used in current cryptographic standard known as DES. We provide an exhaustive experimental analysis of the proposed CA-based S-box in terms of non-linearity, autocorrelation, balance and strict avalanche criterion, and compare it with DES S-boxes. We show that the proposed CA-based S-box has cryptographic properties comparable or better than classical S-box tables. The interesting feature of the proposed S-box is a dynamic flexible structure fully functionally realized by CA, while the classical DES S-box is represented by predefined unchangeable table structure.

Keywords: Cellular Automata, S-boxes, Block Cipher, Cryptography, Boolean Functions.

1 Introduction

Cryptography plays an important role in security of data in the modern world. Two main cryptography approaches are used today to provide a secure communication: secret key and public key systems. An extensive overview of currently known or emerging cryptography techniques used in both types of systems can be found in [8]. The main concerns of this paper are cryptosystems with a secret key. The main interests of this work are CA and their application to design S-boxes. S-boxes functionally realize some Boolean functions, important from point of view of requested cryptographic features performed by S-boxes in secret key systems.

Many known secure standards of symmetric key cryptography, such as, e.g. [3], [4], use efficient and secure algorithms working on the base of S-boxes. S-boxes are ones of the most important components of block ciphers, which are permanently upgraded, or substituted by new better constructions.

In the next section the concept of the S-box and its most known applications in DES cryptographic standards are presented. Section 3 describes the main cryptographic criteria to examine Boolean functions. In section 4 two different Boolean functions are proposed as measures of non-linearity, autocorrelation and balance of S-boxes. Section 5 outlines the concept of CA. In section 6 the idea of creating CA-based S-boxes is proposed. Section 7 presents results of examination of cryptographic features of CA-based S-boxes and their comparison with classical DES S-boxes. The last section concludes the paper.

2 S-Boxes in Cryptography

S-box (see, [3]) is a function $f : B^n \rightarrow B^k$, which from each of n Boolean input values of B^n block consisting of n bits b_i $(i \leq n)$ generates some k Boolean output values called B^k block consisting of k bits b_j $(j \leq k$ and $k \leq n)$, what corresponds to the mapping bit strings $(b_0, b_1, ..., b_n) \rightarrow (b_0, b_1, ..., b_k)$.

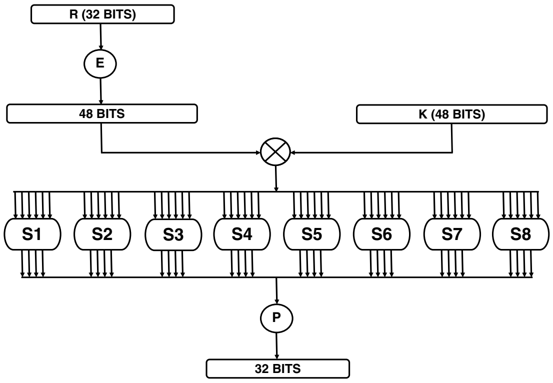


Fig. 1. Ciphering with use of S-boxes *S1, ..., S8* in DES algorithm [3]

One of well known application of S-boxes is using them in Data Encryption Standard (DES) as the "heart" of this algorithm [3]. In DES algorithm 64 input bits are changed by Initial Permutation. After that the 64-bit block is transformed into two blocks of bits composed of 32 bits. One of these two blocks is the block R (see, Fig. 1). The next operation in the algorithm, named E, takes a block of 32 bits as input and yields a block of 48 bits as output. The operation $\otimes$ (XOR: bit-by-bit addition modulo 2) creates from the block $E(R)$ and the 48-bit block of key K a new block of bits (see, Fig. 1). In the next step, 48 bits $(E(R) \otimes K)$ are cut into eight blocks composed of 6 bits each, which are sent to eight S-boxes *S1, ..., S8*. Each of DES S-boxes is the function, which map *6* input bits into *4* output bits. Reassuming, these eight wide known functions collectively transform the 48-bit input block into 32-bit output block (see, Fig. 1).

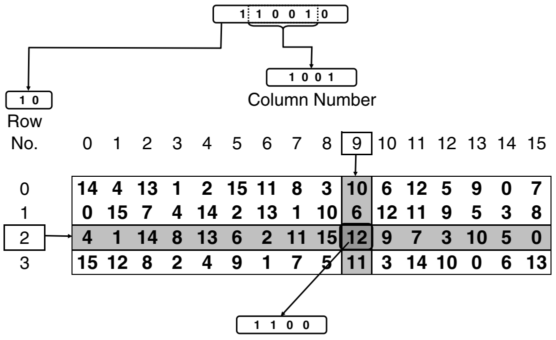


Fig. 2. Mapping the 6 bits into the 4 bits with use of S-box S1 (in DES algorithm [3]) represented as table

Each of the functions *S1, ..., S8* are tables composed of 16-columns and 4-rows. Each function takes a 6-bit block as input and yields a 4-bit block as output.

Let us consider the function *S1* represented in Fig. 2 by specially designed table. Suppose that the input block of this function is the block B^6, e.g. *110010.* Two bits from B^6, the first and the last one (bits *10*) define row *2* of the *S1* block. Four middle bits *1001* define the column *9* of the *S1* block. Intersection of the column *9* and row *2* points in the table the number *12*, what corresponds to *1100*, and these bits are considered as the B^4 output block.

S-boxes are also used in modern symmetric key cryptography systems, e.g. in the new standard Advanced Encryption Standard (AES) [4], successor of DES.

Let us note that the classical S-boxes, as described above, are constant, not flexible structures requesting predefined sizes of memory. Therefore, it is hard to use them in new designed cryptographic algorithms, which request using dynamic S-boxes. The purpose of this study was to design a flexible S-boxes, ready to use in cryptographic algorithms with dynamic S-boxes. It seems that CA are appropriate tool to design such S-boxes.

3 Cryptographic Criteria for Evaluation of Boolean Functions

A motivation for applying CA to realize S-boxes steams from potentially very interesting features of CA. CA have a computational possibilities equivalent to Universal Turing Machine [12], what means that such Boolean functions can be realized. What more, CA of a given size and with their rules (see, Section 4) can potentially realize not one, but a number of S-box functions, what gives a possibility of designing much more stronger cryptography systems. The important issue is also efficiency of running cryptography systems. CA is a highly parallel system, easy in hardware implementation, what results in high efficiency of CA-based systems.

The quality of S-boxes designed with use of CA must be verified by required properties of S-boxes. The most important definitions and dependencies related to this issue are recalled below from cryptographic literature [1], [2], [10], [13].

A Boolean function $f : Z_2^n \rightarrow Z_2$ maps n binary inputs to a single binary output. The list of the size of 2^n of all possible outputs is the *truth table. Polarity* form of *the truth table* is denoted by $\hat{f}(x)$ and defined as:

$$\hat{f}(x) = (-1)^{f(x)}. \tag{1}$$

Boolean function is named a linear function, when it can be expressed as a *XOR* function defined on input variables. Let $x = (x_1, x_2, ..., x_n)$ be input variables, then the linear function $L_\omega(x)$ defined with use of coefficients $\omega \in Z_2^n$ is expressed by the equation:

$$L_\omega(x) = \omega_1 x_1 \otimes \omega_2 x_2 \otimes ... \otimes \omega_n x_n, \tag{2}$$

where $\omega_i x_i$ denotes *AND* operation on i-th bit of ω and x, the operation $\otimes$ denotes *XOR* on bits. A set of *affine functions* is the set composed of linear functions and its complements.

Walsh Hadamard Transform $\hat{F}_f(\omega)$ defines a correlation between a function f and relevant linear function $L_\omega(x)$. It measures how well the linear function approximates function f. Walsh Hadamard Transform is a product of polar forms f and L_ω and can be expressed as:

$$\hat{F}_f(\omega) = \sum_{x \in B^n} \hat{f}(x)\hat{L}_\omega(x). \tag{3}$$

The absolute maximum value in the space of transforms is defined by:

$$WH_{max}(f) = max_{\omega \in B^n} |\hat{F}_f(\omega)|. \tag{4}$$

The non-linearity N_f of a Boolean function f is the minimal distance of the function f to the set of affine functions and is calculated as:

$$N_f = \frac{1}{2}(2^n - WH_{max}(f)). \tag{5}$$

The higher is the non-linearity of observed ciphers (WH_{max} is low) the cipher is more difficult to cryptanalysis.

The next important property of ciphers is autocorrelation AC_f. Autocorrelation defines correlation between polar form $f(x)$ and its polar *shifted version*, $f(x \otimes s)$. Autocorrelation of a Boolean function f is defined by Autocorrelation Transform given by the equation:

$$\hat{r}_f(s) = \sum_x \hat{f}(x)\hat{f}(x \otimes s), \tag{6}$$

where $s \in Z_2^n - \{0\}$. The absolute maximum value of any autocorrelations is denoted by the equation:

$$AC_f = max_{s \neq 0} |\sum_x \hat{f}(x)\hat{f}(x \otimes s)|. \tag{7}$$

The lowest is the autocorrelation of observed ciphers the cipher is more difficult to attacks.

Balance (regularity) is another important criterion which should be fulfilled by a Boolean function used in ciphering (see, [13]). This means that each output bit *(0 or 1)* should appear an equally number of times for all possible values of inputs. The balance of a Boolean function is measured using its Hamming Weight, and is defined as:

$$HW = \frac{1}{2}(2^n - \sum_{x \in B^n} \hat{f}(x)). \tag{8}$$

Boolean function is balanced when its Hamming Weight is equal to 2^{n-1}.

Strict Avalanche Criterion (SAC) was first introduced by Webster and Tavares [10]. A Boolean function of n variables satisfy SAC, if complements of any of the n input bits result in changing the output bit with probability equal to $\frac{1}{2}$. It means, that for each of n-element vector c^n with only one the $i-th$ bit of this vector equal to 1 (c_i^n) the following equation is satisfied:

$$\sum_{x \in B^n} f(x) \otimes f(x \otimes c_i^n) = 2^{n-1}. \tag{9}$$

The analysis of satisfaction of SAC for Boolean function f is measured by the distance $dSAC$, which is expressed by the equation:

$$dSAC_f = max_{1 \leq i \leq n} |2^{n-1} - \sum_{x \in B^n} f(x) \otimes f(x \otimes c_i^n)|. \tag{10}$$

One can see that for ideally balanced a Boolean function f the value of $dSAC$ is equal to 0. For the function not ideally balanced the values of $dSAC$ will be in the range $(0, 2^{n-1}]$.

4 Measuring Cryptographic Properties of S-Boxes

S-boxes as functions mapping n input bits into k output bits under condition $k \leq n$ generally do not satisfy conditions to be a Boolean function, because the number of output bits of S-boxes is usually higher then one bit ($1 \leq k$). However, the quality of block ciphers received with use of S-boxes is usually measured by criteria proper to Boolean functions. The question which arises is how to apply these criteria to S-boxes (block ciphers). Let us consider two possible methods to solve this problem.

4.1 Method 1: Linear Combination of Single-Output S-Boxes

S-boxes are functions, which from n input bits generate k output bits. However, a Boolean function returns as output one bit. To use Boolean functions criteria to examine S-boxes, we need to transform all k bits output of an S-box into one output bit. After this modification, we obtain a new Boolean function which can be defined as: $f_\beta : B^n \rightarrow B^1$, and expressed by the formula (see, [1], [2], [5], [7]):

$$f_\beta(x) = \beta_1 f_1(x) \otimes \beta_2 f_2(x) \otimes ... \otimes \beta_k f_k(x). \tag{11}$$

The new function is a linear combination of k functions $f_i(x)$, $i \leq k$, where $\beta_i \in B^k$. Each of these functions is defined as a simple S-box (single-output S-box, a part of the $n \times k$ S-box). The relationship (vector $(\beta_1, ..., \beta_k)$) between simple S-boxes is a result of the S-box table composition.

Under this approach cryptographical properties of S-boxes presented in section 3 are calculated with use of the Boolean function $f_\beta(x)$.

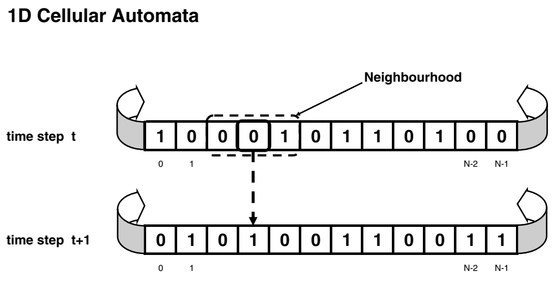
4.2 Method 2: Set of Single-Output S-Boxes

In this method the S-box is considered as a set of simple S-boxes. A simple S-box satisfies conditions to be a Boolean function. Each simple S-box is a function: $f_i : B^n \rightarrow B^1$, where $i = 1, 2, ..., k$ and the 1-bit output is one of k output bits of the S-box.

Cryptographic properties of an S-box are measured under this method separately for each simple S-box ($\{f_1, ..., f_k\}$), where k is the number of output bits in the $n \times k$ S-box. To analyze the $n \times k$ S-box, using the single-output S-box method, the k single-output S-boxes are considered. It results in analyzing the k Boolean functions corresponding to single-output S-boxes and evaluating their non-linearities, autocorrelations, balances and SACs. Partial results are compared and the worst ones become the final evaluation of the analyzed S-box. Such an approach was used in cryptanalysis and recently in [6] to analyze the 6×6 S-boxes.

5 The Concept of Cellular Automata

One dimensional (1D) CA is in the simplest case a collection of two-state elementary cells arranged in a lattice of the length N, and locally interacting in a discrete time t. For each cell i called a central cell, a neighbourhood of a radius r is defined, consisting of $n_i = 2r + 1$ cells, including the cell i. When considering a finite size of CA, and a cyclic boundary condition is applied, it results in a circle grid (Fig. 3). It is assumed that a state q_i^{t+1} of a cell i at the time $t + 1$ depends only on states of its neighbourhood at the time t, i.e. $q_i^{t+1} = f(q_i^t, q_{i1}^t, q_{i2}^t, , q_{in}^t)$, and a transition function f, called a rule, which defines a rule of updating state of the cell i (Fig. 3). A length L of a rule and a number of neighbourhood states for a binary uniform CA is $L = 2^n$, where $n = n_i$ is a number of cells of a given neighbourhood, and a number of such rules can be expressed as 2^L. Fig. 3 presents an example of the rule 01011010 (called also rule 90) for $r = 1$. The length L of the rule consists of 8 bits. CA for systems with a secrete key were first studied by Wolfram [11]. He used 1D uniform CA to generate pseudorandom numbers. 1D uniform CA use only one rule as transition function, in opposite to 1D nonuniform CA, which use more than one rule to update cells of CA.



Rule of CA

Neighbourhood radius r=1, rule $01011010_2 = 90_{10}$

Neighbourhood state	111	110	101	100	011	010	001	000
Rule	0	1	0	1	1	0	1	0

Fig. 3. 1D cellular automata with neighbourhood radius equal to 1

6 Constructing CA-Based S-Boxes

6.1 Major Principles

A classic S-box is a function expressed as a table containing natural numbers. Cryptographic literature shows many examples and methods of searching S-box tables. The quality of S-boxes are measured with use of different functions which examine their different properties [5], [1], [7], [10], [13]. Some of the most important test functions were presented in section 3. In [5], [1], [7] authors treat the problem of designing S-box tables as a combinatorial optimization problem and apply different metaheuristics to search solutions in the huge space of S-box tables solutions. Recently [9] we have proposed CA-based approach to create S-boxes in the form not tables, but some virtual entities.

The CA-based S-box can be seen as CA composed of the following elements:

– a number of CA cells performing the role of background
– a number of CA cells performing the role of input/output of CA-based S-box
– an initial state of CA
– an appropriate rule/rules of CA.

It is assumed that CA will evolve during a number of time steps. Selected cells of CA (in its initial state) serve as input bits of the S-box, and the same cells, after declared time steps, are considered as the output of the S-box. To construct CA performing the S-box function it is necessary to find appropriate CA rules and verify produced results according to the S-box functions criteria.

6.2 Details of Construction

The first step in constructing the $n \times k$ CA-based S-box is selecting a number of CA cells. A number of CA cells must be not lower than $max|n, k|$. This number should be also enough large to generate cycles longer than the number of CA time steps [6]. It is worth to say at this moment, about inputs and outputs of CA-based S-box. Construction of $n \times n$ S-box is simple (see, [9]), because we can use n cells of CA and from n inputs we obtain n outputs. How to use CA to construct $n \times k$ S-boxes, when $n \geq k$? For this purpose we propose to consider the first n CA cells at the time step $t = 0$ as input cells, and the first k CA cells at the last time step as output cells (see, Fig. 4).

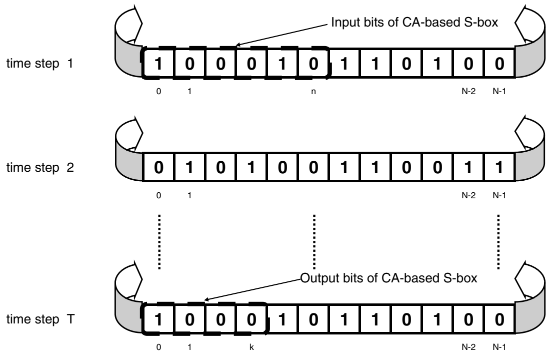


Fig. 4. CA-based the $n \times k$ S-box construction

In [9] we used quite long CA (with a number CA cells = 100), where significant input/output-bit cells collectively cooperate with other cells called the background. This construction is large enough to satisfy condition for non-cycle construction. The number of time steps which satisfy this condition is equal to 100 (see, [9]).

Not every CA rule is suitable to provide proper quality for CA-based S-box. We selected four rules $\{30, 86, 135, 149\}$ (for CA, with neighborhood radius $r = 1$) as only proper for this purpose (see also, [9]). These rules change CA cells in time step t into cells in time step $t + 1$, as follows:

$$Rule\ \ 30: \ q_i^{t+1} = q_{i-1}^t \otimes (q_i^t \vee q_{i+1}^t), \tag{12}$$

$$Rule\ \ 86: \ q_i^{t+1} = (q_{i-1}^t \vee q_i^t) \otimes q_{i+1}^t, \tag{13}$$

$$Rule\ 135: \ q_i^{t+1} = \overline{q_{i-1}^t \otimes (q_i^t \vee q_{i+1}^t)}, \tag{14}$$

$$Rule\ 149: \ q_i^{t+1} = \overline{(q_{i-1}^t \vee q_i^t) \otimes q_{i+1}^t}. \tag{15}$$

Other single rules are too weak to be used in CA-based S-boxes.

The initial state of CA (the first n bits interpreted as the S-box input) is randomly set and CA starts to run. After a predefined number of time steps the CA stops and its the first k bits are treated as output bits, which are next used to evaluate quality of CA-based S-boxes. All experimental results presented in the next section are calculated assuming the start of CA from many different initial states.

7 Analysis and Comparison of S-Boxes

7.1 DES S-Boxes Analysis

In [9] we proposed the 8×8 CA-based S-boxes, which offer cryptographic quality in general comparable or better then nowadays constructed S-box tables. In this paper we propose a CA-based construction of DES S-boxes. They perform mapping from the 6 bits to the 4 bits (the 6×4 S-boxes). The length of the CA is equal to 100 cells. The CA is controlled by one of rules from the set presented in section 6.2. The CA will evolve during 100 time steps. As it was shown in section 6.2, in the corresponding CA the first 6 input bits are considered as input of DES S-box and, after evolving CA a predefined number of steps, the first 4 bits are considered as output bits.

Table 1. The range of values and the best theoretical values of cryptographical properties (N_f, AC_f, HW_f and $dSAC_f$) for the 6×4 S-boxes

	N_f	AC_f	HW_f	$dSAC_f$
The best value	32	0	32	0
Range of values	[0, 32]	[0, 64]	[0, 32]	[0, 32]

Table 2. Non-linearity (N_f), autocorrelation (AC_f), balance (HW_f) and distance to Strict Avalanche Criterion ($dSAC$) for DES 6×4 S-boxes (Method 1 - linear combination of simple S-boxes, Method 2 - set of simple S-boxes)

S-box	Method 1	Method 2
	(N_f, AC_f, HW_f, $dSAC_f$)	(N_f, AC_f, HW_f, $dSAC_f$)
DES, S1	(14, 48, 32, 24)	(18, 40, 32, 16)
DES, S2	(20, 48, 32, 16)	(18, 56, 32, 28)
DES, S3	(16, 40, 32, 12)	(18, 48, 32, 24)
DES, S4	(16, 64, 32, 32)	(22, 24, 32, 12)
DES, S5	(12, 40, 32, 16)	(18, 40, 32, 20)
DES, S6	(20, 40, 32, 8)	(20, 48, 32, 20)
DES, S7	(18, 32, 32, 16)	(14, 48, 32, 24)
DES, S8	(16, 48, 32, 24)	(20, 40, 32, 20)

Let us analyze cryptographic quality of DES S-boxes and compare it with quality of the new proposed CA-based S-boxes performing the same role as DES S-boxes. For this purpose we will use measures presented in section 4.

The results of analysis of DES S-boxes are presented in Table 1 and Table 2. Table 1 shows the best (ideal) values and possible ranges of values of cryptographical properties of the the 6×4 S-boxes independently on their construction.

Table 2 shows values of non-linearity, autocorrelation, balance and distance to strict avalanche criterion for all S-boxes $S1, S2, ..., S8$ in DES. These results were obtained with use of two methods (Method 1, Method 2) of construction of Boolean function for S-boxes presented in section 4.

One can see that while the ideal value of non-linearity is equal to 32, this value for DES S-boxes changes in the range $[12, 20]$ (Method 1) and $[14, 22]$ (Method 2). The values of autocorrelation change in the range $[32, 64]$ (Method 1) and $[24, 56]$ (Method 2). The value of balance is equal to 32 and is independent on the method. The values of distance to strict avalanche criterion ranged $[8, 32]$ (Method 1) and $[12, 28]$ (Method 2). We can see that values of non-linearity, autocorrelation and distance to SAC are quite far from ideal ones. The only value of balance has the ideal value.

7.2 Analysis of CA-Based S-Boxes Corresponding to DES S-Boxes

The results of analysis of CA-based S-boxes with rules $30, 86, 135, 149$ are presented in Table 3. These results were obtained on the base of 10000 runs from random initial CA states. It is worth to notice the main difference between running DES S-boxes and CA-based S-boxes. For each input of a S-box we obtain one output in DES S-boxes, while for a single input of CA-based S-boxes we can obtain a number of outputs, which depends on the number of CA initial states. Therefore, the Table 3 shows results related to extreme, the best and the worst CA-based S-boxes (values of non-linearity, autocorrelation, Hamming Weight and distance to fulfills SAC). More exactly, it contains fours, where either non-linearity, autocorrelation, Hamming Weight or distance to fulfill SAC takes minimal or maximal value.

One can see that there exist initial states of CA which provide values of N_f, AC_f and $dSAC_f$ better than corresponding values of DES S-boxes, and this is true for each of considered CA rules. For example, for rule 30, 135, 149 the four $(24, 16, 32, 4)$ and for rule 86 the four $(24, 16, 32, 8)$ is much better then any four for DES S-boxes shown in Table 2, measured by Method 1 - linear combination of simple S-boxes. For Method 2 - set of simple S-boxes, CA-based S-boxes are comparable with DES S-box $S4$ and better then other DES S-boxes (compare third column in Table 2 with second in Table 3).

7.3 Analysis of Non-linearity of CA-Based S-Boxes

Let us analyze schedule of non-linearity, autocorrelation, Hamming Weight and distance to SAC for CA-based S-boxes. Fig. 5 presents detailed study of non-linearity in CA-based the 6×4 S-boxes with use of Method 1 (a) and Method 2 (b). It shows percentage distribution of CA corresponding to different initial states and their quality in the sense of non-linearity. Single CA were presented on X-axis of Fig. 5 in increasing order of N_f (from left to right). One can see that there exist a relatively large number of initial CA states which provide good values of N_f (and better than DES S-boxes) see, range (21, 25) for Method 1 (see, Fig. 5a), and few (value 23) for Method 2 (see, Fig. 5b).

Table 3. Properties of non-linearity (N_f), autocorrelation (AC_f), balance expressed by Hamming Weight (HW_f) and distance to fulfills SAC ($dSAC_f$) for the 6×4 CA-based S-boxes selected from 10000 initial states (the best/worst values in bold)

Method 1 - linear combination of simple S-boxes

S-box CA rule:	The best CA-based S-boxes (N_f, AC_f, HW_f, $dSAC_f$)	The worst CA-based S-boxes (N_f, AC_f, HW_f, $dSAC_f$)
30	(**25**, 20, 31, 6), (23, **12**, 31, 6), (24, 16, **32**, 4), (25, 28, 29, **2**)	(**14**, 40, 26, 16), (19, **52**, 31, 10), (17, 36, **17**, 10), (21, 44, 29, **22**)
86	(**25**, **12**, 31, 6), (24, 16, **32**, 8), (22, 24, 30, **0**)	(**13**, 36, 31, 14), (22, **48**, 28, **24**), (17, 28, **17**, 14)
135	(**25**, **12**, 29, 6), (24, 16, **32**, 4), (22, 24, 26, **0**)	(**13**, 36, 31, 18), (19, **52**, 31, 10), (18, 48, **18**, 16), (20, 48, 32, **24**)
149	(**25**, 20, 31, 2), (24, **16**, 28, **0**), (24, **16**, **32**, 4)	(**14**, 48, 30, 12), (19, **52**, 23, 6), (17, 28, **17**, 14), (19, 44, 31, **22**)

Method 2 - set of simple S-boxes

S-box CA rule:	The best CA-based S-boxes (N_f, AC_f, HW_f, $dSAC_f$)	The worst CA-based S-boxes (N_f, AC_f, HW_f, $dSAC_f$)
30	(**23**, **20**, 27, 10), (**23**, 28, **31**, 10), (**23**, 28, 27, **6**)	(**13**, 44, 30, 14), (20, **56**, 25, 12), (17, 36, **17**, 18), (18, 48, 28, **24**)
86	(**23**, 28, 25, 6), (21, **20**, 25, 10), (17, 36, **31**, 14), (22, 32, 22, **4**)	(**12**, 40, 26, 20), (17, **52**, 24, 18), (15, 44, **15**, **22**)
135	(**23**, **20**, 29, 10), (22, 32, **32**, 8), (22, 40, 28, **4**)	(**12**, 40, 27, 16), (20, **56**, 24, 16), (16, 32, **16**, 12), (15, 52, 29, **26**)
149	(**23**, **20**, 27, **6**), (20, 32, **32**, 12)	(**12**, 36, 29, 16), (19, **52**, 27, **26**), (16, 48, **16**, 12)

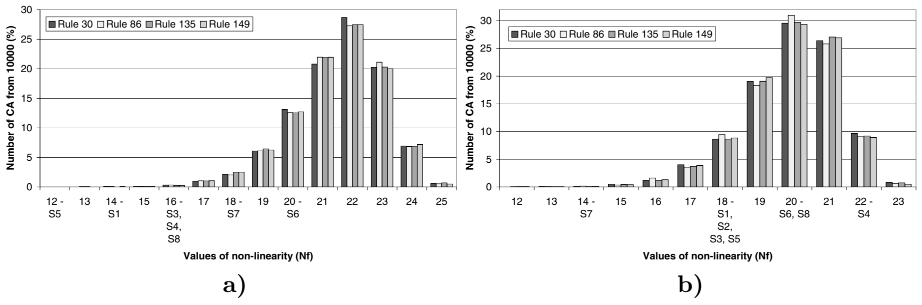


Fig. 5. Percentage of CA corresponding to N_f for CA-based S-boxes, measured by Method 1 (a) and Method 2 (b)

As it was mentioned earlier, each CA with a given initial state can be considered as an independent CA-based S-box. Table 4 presents the percentage of CA for a given initial state and selected CA rules, which give results of non-linearity better than DES S-box tables. On the base of data presented in Table 4, we

Table 4. Percentage of CA corresponding to CA-based S-boxes characterizing by values of N_f better than DES S-boxes (Method 1 - linear combination of simple S-boxes, Method 2 - set of simple S-boxes)

Rule	30	86	135	149
CA better than the best DES S-box (Method 1)	77.22%	77.75%	77.15%	77.09%
CA better than the best DES S-box (Method 2)	0.8%	0.65%	0.74%	0.5%

can conclude this part of study, that almost 80% randomly selected CA-based S-boxes (measured by Method 1) gives better results than DES S-boxes, but almost 1% selected CA-based S-boxes (measured by Method 2) give better results than DES S-boxes, for N_f.

7.4 Analysis of Autocorrelation of CA-Based S-Boxes

Fig. 6 presents detailed study of autocorrelation in CA-based the 6×4 S-boxes with use of Method 1 (a) and Method 2 (b). It shows percentage distribution of CA corresponding to different initial states and their quality in the sense of autocorrelation. Single CA were presented on X-axis of Fig. 6 in increasing order of AC_f (from left to right). One can see that there exist a relatively large number of initial CA states which provide good values of AC_f (and better than DES S-boxes) see, range (12, 28) for Method 1 (see, Fig. 6a), and few (value 20) for Method 2 (see, Fig. 6b).

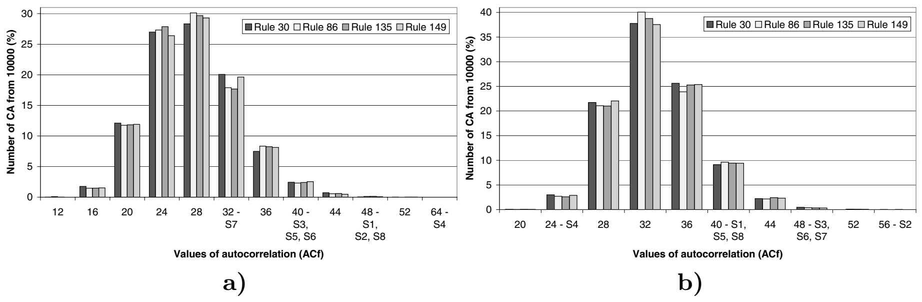


Fig. 6. Percentage of CA corresponding to AC_f for CA-based S-boxes, measured by Method 1 (a) and Method 2 (b)

Table 5 presents the percentage of CA for a given initial state and randomly selected CA rules, which give results of autocorrelation better than DES S-box tables. On the base of data presented in Table 5, we can conclude this part of study, that 70% randomly selected CA-based S-boxes (measured by Method 1) gives better results than DES S-boxes, but less than 1% selected CA-based S-boxes (measured by Method 2) give better results than DES S-boxes, for AC_f.

Table 5. Percentage of CA corresponding to CA-based S-boxes characterizing by values of AC_f better than DES S-boxes (Method 1 - linear combination of simple S-boxes, Method 2 - set of simple S-boxes)

Rule	30	86	135	149
CA better than the best DES S-box (Method 1)	69.2%	70.75%	70.79%	69.12%
CA better than the best DES S-box (Method 2)	0.04%	0.02%	0.04%	0.03%

7.5 Analysis of Balance of CA-Based S-Boxes

DES S-boxes are always balanced, because despite of the method of measure provide an output with the same number of 0s and 1s ($HW_f = 32$).

For all CA (with their different initial states), CA-based S-boxes do not provide the same number of 0s and 1s for selected output cells of CA. As a measure of CA-based S-boxes balance, values of HW_f were calculated.

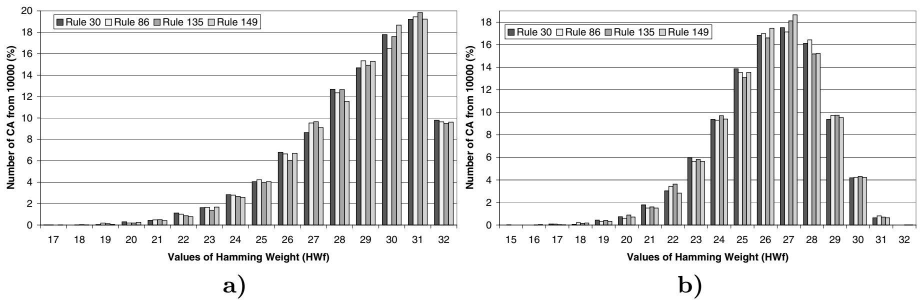


Fig. 7. Percentage of CA corresponding to balance (expressed by Hamming Weight) of values for CA-based S-boxes, measured by Method 1 (a) and Method 2 (b)

Histograms in Fig. 7 present Hamming Weight of balance for CA-based S-boxes in 10000 CA (with different, random initial states). A number of balanced CA-based S-boxes (with $HW_f = 32$) is almost 10% (see, Fig. 7a) and less than 1% (see, Fig. 7b), for Method 1 and Method 2, respectively. However, for CA we need to analyze this problem more widely. Balance of CA, should be calculated from a number of CA with random initial states, because CA is not one initial state, but large number of CA with all of possible initial states. In our case, CA size is equal to 100. A number of possible initial states (and different CA) is equal to 2^{100}. As a representative sample was used 10000 CA with random initial states. For each rule, from 10000 CA, number of 0s and 1s for selected output cells of CAs was averaged and HW_f was calculated. Obtained Hamming Weights was near to value 32 (ideal), what characterizes CA-based S-boxes properly.

7.6 Analysis of SAC of CA-Based S-Boxes

DES S-boxes not satisfy SAC. These S-boxes are characterized by high values of dSAC, except of S-box S6 measured by the Method 1 and S4 measured by the Method 2 (see, Table 2). The distance to SAC for S-box S6 is the shortest and equal to 8 (Method 1), also for S4 is equal to 12 (Method 2).

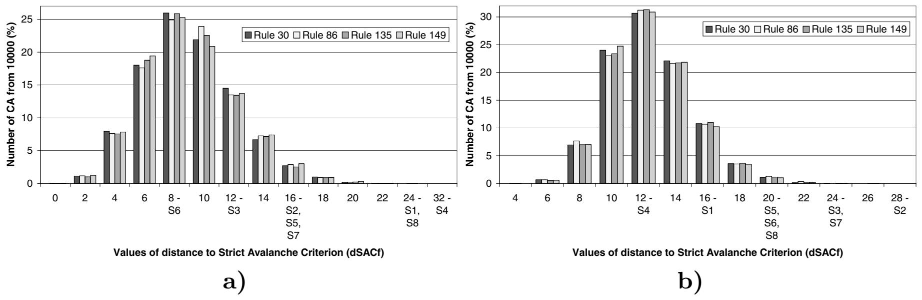


Fig. 8. Percentage of CA corresponding to $dSAC$ values for CA-based S-boxes, measured by Method 1 (a) and Method 2 (b)

Fig. 8 presents histograms of $dSAC$ for 10000 CA (with different, random initial states). One can observe that $dSAC$ for CA-based S-boxes is in the range $[0, 24]$ for the Method 1, and $[4, 26]$ for the Method 2. For the best CA-based S-boxes $dSAC$ are equal 0 for rule 86, 135, 149 and 2 for 30 (Method 1), similarly value 4 for rules 86, 135 and 6 for rules 30, 149 (Method 2). It is much better (shorter distance) then for DES S-Boxes. For the Method 1, the most frequently obtained $dSAC$ values in CA are the following: $6, 8, 10$. For the Method 2, $dSAC$ values, which were obtained the most frequent in CA are values $10, 12, 14$.

Table 6. Percentage of CA corresponding to CA-based S-boxes characterizing by values of $dSAC_f$ better than DES S-boxes (Method 1 - linear combination of simple S-boxes, Method 2 - set of simple S-boxes)

Rule	30	86	135	149
CA better than the best DES S-box (Method 1)	27.1%	26.4%	27.34%	28.54%
CA better than the best DES S-box (Method 2)	31.61%	31.35%	30.91%	32.31%

When we interpret each CA with an initial state as independent CA-based S-box, than we can conclude this part of the study, that more than 26% (Method 1, see, Table 6) and 31% (Method 2, see, Table 6) selected CA-based S-boxes give shorter distances to SAC than DES S-boxes.

8 Conclusions and Future Work

The paper presents an idea of creating S-boxes using CA-based approach. Classical S-boxes based on tables are fixed structure constructions. We are interested in creating CA-based S-boxes, which are dynamical structures. CA from input block of bits generates output block of bits and is evaluated by the same examine criteria like the traditional S-box. Conducted experiments have shown that the 6×4 CA-based S-boxes characterized in most, by a high non-linearity and low autocorrelation independent on method of its measure. These values in many cases are better than classical tables of DES S-boxes. Balance of proposed S-boxes seems to be quite good for single CA, but when we calculates Hamming Weight from each CA, CAs show us almost balanced. The average value of $dSAC$ of CA-based S-boxes are comparable with the best DES S-boxes, but for many single CA (single CA-based S-boxes) $dSAC$ values are much more low (than better), than for DES tables. The next step of researches will be examining all of possible initial states of CA under conditions of the highest possible N_f, the lowest AC_f, perfect balance (50%) (these means $HW_f = 32$), and also for the lowest value of $dSAC_f$.

References

1. Clark, J.A., Jacob, J.L., Stepney, S.: The Design of S-Boxes by Simulated Annealing. New Generation Computing 23(3), 219–231 (2005)
2. Dowson, E., Millan, W., Simpson, L.: Designing Boolean Functions for Cryptographic Applications, Contributions to General Algebra 12, pp. 1–22. Verlag Johannes Heyn, Klagenfurt (2000)
3. Federal Information Processing Standards Publication, FIPS PUB 46-3, DES (1999), http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf
4. Federal Information Processing Standards Publications, FIPS PUBS 197, AES (2001), http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
5. Millan, W., Burnett, L., Carter, G., Clark, A., Dawson, E.: Evolutionary Heuristics for Finding Cryptographically Strong S-Boxes. In: Varadharajan, V., Mu, Y. (eds.) ICICS 1999. LNCS, vol. 1726, pp. 263–274. Springer, Heidelberg (1999)
6. Mukhopadhyay, D., Chowdhury, D.R., Rebeiro, C.: Theory of Composing Non-linear Machines with Predictable Cyclic Structures. In: Umeo, H., Morishita, S., Nishinari, K., Komatsuzaki, T., Bandini, S. (eds.) ACRI 2008. LNCS, vol. 5191, pp. 210–219. Springer, Heidelberg (2008)
7. Nedjah, N., de Macedo Mourelle, L.: Designing Substitution Boxes for Secure Ciphers. International Journal Innovative Computing and Application 1(1), 86–91 (2007)
8. Scheier, B.: Applied Cryptography. Wiley, New York (1996)
9. Szaban, M., Seredynski, F.: Cryptographically Strong S-Boxes Based on Cellular Automata. In: Umeo, H., Morishita, S., Nishinari, K., Komatsuzaki, T., Bandini, S. (eds.) ACRI 2008. LNCS, vol. 5191, pp. 478–485. Springer, Heidelberg (2008)
10. Webster, A.F., Tavares, S.: On the Design of S-Boxes. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 523–534. Springer, Heidelberg (1986)
11. Wolfram, S.: Cryptography with Cellular Automata. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 429–432. Springer, Heidelberg (1986)
12. Wolfram, S.: A New Kind of Science. Wolfram Media Inc., Champaign (2002)
13. Youssef, A., Tavares, S.: Resistance of Balanced S-boxes to Linear and Differential Cryptanalysis. Information Processing Letters 56, 249–252 (1995)

Hierarchical Dependency Graphs: Abstraction and Methodology for Mapping Systolic Array Designs to Multicore Processors[⋆]

Sudhir Vinjamuri and Viktor Prasanna

3740 McClintock Avenue EEB 200, Ming Hsieh Department of Electrical Engineering
University of Southern California, California, USA 90089-2562
Tel.: +1-213-740-1521, +1-213-740-4483; Fax: +1-213-740-4418
sudhir.vinjamuri@usc.edu, prasanna@usc.edu

Abstract. Systolic array designs and dependency graphs are some of the most important class of algorithms in several scientific computing areas. In this paper, we first propose an abstraction based on the fundamental principles behind designing systolic arrays. Then, based on the abstraction, we propose a methodology to map a dependency graph to a generic multicore processor. Then we present two case studies: Convolution and Transitive Closure, on two state of the art multicore architectures: Intel Xeon and Cell multicore processors, illustrating the ideas in the paper. We achieved scalable results and higher performance compared to standard compiler optimizations and other recent implementations in the case studies. We comment on the performance of the algorithms by taking into consideration the architectural features of the two multicore platforms.

Keywords: parallel programming, multicore, systolic array designs, dependency graphs, high performance computing.

1 Introduction and Background

Signal and image processing algorithms, matrix and linear algebra operations, graph algorithms, molecular dynamics and geo-physics are some of the core scientific computing research areas ([1]). The 70s, 80s saw the upsurge of a revolutionary high performance computing technology - systolic array processors ([3]) necessitated by increasing demands of speed and performance in these areas. A lot of research work has been done to expose the parallelism and pipelining available in several important scientific computing applications to be exploited by systolic array processors ([1], [2], [4], [5]). For many of these algorithms, the

⋆ This research was partially supported by the NSF under grant number CNS-0613376. NSF equipment grant CNS-0454407 is gratefully acknowledged. The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

hardware needed to be flexible and robust to be able to adapt to new problems and also variations in known algorithms. Many solutions were proposed to this problem such as configurable systolic array processing platforms, FPGAs and reconfigurable computing platforms ([8]).

Today's computing revolution is driven by massive on-chip parallelism ([9]). For the foreseeable future, high performance computing machines will almost certainly be equipped with nodes featuring multicore processors where each processor contains several full featured general purpose processing cores, private and shared caches. So the primary motivation of this work is to study, how "classical" algorithms can be "recycled" now that parallel computing has a renaissance with the advent of multicore computers. Also, the configurable platforms of systolic arrays pale out in comparison with multicore processors of comparable area and cost in terms of raw compute power and peak performance achievable due to high clock rate, chip density and economies of scale of multicore processors. Hence, it is highly desirable to extract parallelism and pipelining necessary for systolic array designs from multicore processors. If done intelligently, this will result in highly optimized performance since those properties are inherent to multicore architectures.

To the best of our knowledge, there is no known prior work to map dependency graphs or systolic arrays to the current generation of multicore processors. We believe this is the first attempt for studying this problem. The main challenge of coming up with a methodology to map any systolic array designs to a multicore processor requires deep understanding of systolic array design procedures, data partitioning, scheduling operations and data flow control. On a multicore processor this poses extra challenges where synchronization of the operations and data of the cores has to be controlled by the programmer. The remainder of this paper is organized as follows: Section 2 is the crux of this paper where we present the approach for this study, the abstraction of Hierarchical Dependency Graphs and mapping methodology to multicore processors. In Section 3, we present two case studies of two algorithms on two multicore architectures. We conclude the paper with a brief summary and avenues for future work in Section 4.

2 Hierarchical Dependency Graphs

In this Section, first we substantiate the approach adopted in this paper. Then in Section 2.2, we discuss the properties of dependency graphs, differentiate and define data flows. In Section 2.3, we describe the abstraction of hierarchical dependency graphs and their properties. In Section 2.4, we discuss the steps to generate a mapping for a generic dependency graph to a multicore processor.

2.1 Approach for This Study

Systolic array algorithms may or may not have a specific design methodology/steps ([6]). The methodology for designing systolic arrays is a description of a sequence of steps at best and not a concrete algorithm that takes an application and designs a systolic array for it. Figure 1 shows one of the widely used set of steps to design systolic arrays ([1]).

Dependency graphs are converted into systolic arrays by passing them through a sequence of steps one of which is single assignment code. The motivation of single assignment code is to avoid broadcasting in the VLSI design technology because it brings down the clock rate. But, in the case of today's multicore processors, while write conflict between cores to a memory location is a problem, broadcasting is not a problem currently because all that means is a value being read by all processors which, as will become evident from the case studies, is not a problem. Hence we consider the designs at the level of dependency graphs. But one ambiguity arises in the cases where the systolic array design is very similar or the same as the dependency graph. To resolve this issue, we consider systolic array designs at the level of dependency graphs itself in this paper.

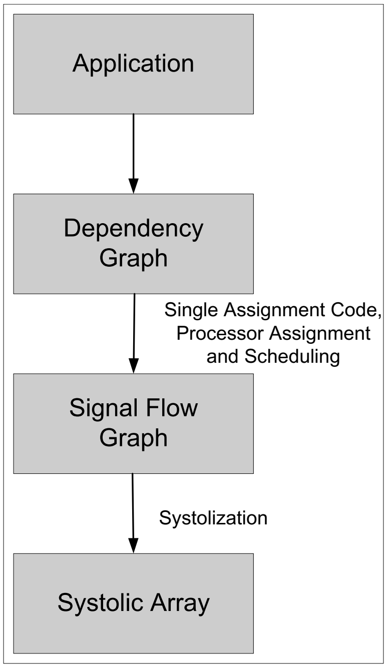


Fig. 1. Steps for designing a systolic array

2.2 Dependency Graphs

The theory of dependency graphs has been discussed in [10]. These were later used ([1]) in the design of systolic arrays. In this paper, to make our approach intuitive, instead of getting into intricate details of the definitions of dependency graphs, we use an example of a dependency graph to describe our ideas.As the properties used in the example are generic to dependency graphs and also through the case studies, it will be clear that our approach can be applied to any dependency graph. We chose the systolic array design for transitive closure([2], [4]) as the example. So we briefly describe the problem and its systolic array design below. Additional details can be found in [2].

Transitive closure is a fundamental problem in a wide variety of fields, most notably network routing and distributed computing. Suppose we have a directed graph G with N vertices and E edges. Transitive closure of the graph involves in computing for each vertex of the graph, the subset of vertices to which it is connected and the shortest distance between them (The 0-1 version of transitive closure only shows if the vertices are connected. We use the generic version which gives the shortest distances also). Given the graph, the adjacency matrix W is a 2 dimensional matrix with elements representing edge weights (eqn. 1).

$$w_{i,j} = \begin{cases} 0 & \text{if } i = j \\ weight\ of\ edge\ connecting\ vertex\ i\ to\ j & \text{if } i \neq j and(i,j) \in E \\ \infty & \text{if } i \neq j and(i,j) \notin E \end{cases} \quad (1)$$

The dependency graph design for transitive closure is as follows:
1. Given a graph with N vertices in the adjacency matrix representation (A), feed the matrix into an NxN systolic array of processing elements (PEs) both row-wise from top and column-wise from left as shown in Figure 2.
2. At each PE (i, j), update the local variable $C_{(i,j)}$ by the following formula:

$$C_{(i,j)} = min(C_{(i,j)}, A_{(i,k)} + A_{(k,j)}) \quad (2)$$

where $A_{(i,k)}$ is the value received from the top and $A_{(k,j)}$ is the value received from the left.
3. If i=k, pass the value $C_{(i,j)}$ down, otherwise pass $A_{(k,j)}$ down. If j=k, pass the value $C_{(i,j)}$ to the right, otherwise pass $A_{(i,k)}$ to the right.
4. Finally, when data elements reach the edge of the matrix, a loop around connection should be made such that $A_{(i,N)}$ passes data to $A_{(i,1)}$ and $A_{(N,j)}$ passes data to $A_{(1,j)}$ (see Figure 2).
5. The above computation results in the transitive closure of the input once all the input elements have been passed through the entire array exactly 3 times. We consider 1 of the 3 cycles of the operation for discussing the properties below.

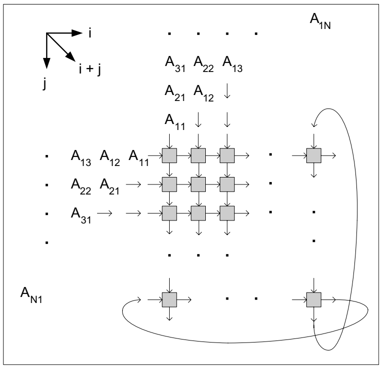


Fig. 2. Systolic Array Implementation of Transitive Closure

Properties of Dependency Graphs: We need two properties of dependency graphs ([1], [10]). We will be using these to explain the properties of hierarchical dependency graphs in Section 2.3.

Property of Parallelism: This property states that, at a specific instant of time, a number of nodes can be processing data in parallel. There are many variations on how this parallelism is present depending on the dependency graph. For e.g. in Figure 2, nodes along the anti-diagonal can process data in parallel. Calling the top right corner node as (0, 0) and $\hat{i}$, $\hat{j}$ axis as shown in the Figure, nodes (1, 0) and (0, 1) can process data in parallel. Similarly, nodes in each set [(2, 0), (1, 1), (0, 2)], [(3, 0), (2, 1), (1, 2), (0, 3)] etc. can process data in parallel.

Property of Modularity, Regularity and Scaling: The array consists of modular processing units with homogeneous interconnections. Moreover, the computing network may be extended indefinitely based on the problem size. For e.g. in Figure 2, the size of the dependency graph is the size of the adjacency matrix. Hence, for a problem of size of N, the dependency graph is of size NxN.

A problem of size $2N$ has a dependency graph of size $2N$x$2N$, which can interpreted modularly as connecting four dependency graphs of a problem of size N (i.e. adjacency matrices of size NxN).

Data Flow: We differentiate the data flowing in the dependency graph into two types:

– Data that is updated at a node before being sent to the next node.
– Data that is sent as it is, without any modifications to the next node.

Definitions

Update Direction (UD): We define UD as the direction in which data that is updated at a node, is sent to another node. In other words, this is the direction in which there is a dependency in data flow. The UD is represented by a unit vector along that direction. There can be more than one UD for a dependency graph. For instance, transitive closure (Figure 2) has two UDs: along $\hat{\imath}$ and $\hat{\jmath}$ directions (i.e. data updated at a node is sent to the neighboring nodes along $\hat{\imath}$ and $\hat{\jmath}$ directions) (These UDs will be used to cut the dependency graphs during the mapping process (discussed in the Section 2.4)).

Unified Update Direction (UUD): In the case where a dependency graph has multiple UDs, we define UUD as the unit vector along the average of the UDs. Every dependency graph whether it has one UD or multiple UDs, has only a single UUD. In the case where the dependency graph has only a single UD, that itself will become the UUD. So the UUD for transitive closure is along $(\hat{\imath}+\hat{\jmath})/\sqrt{2}$ (The UUD will be used in scheduling of the hierarchical dependency graph during the mapping process (discussed in the Section 2.4)).

Θ **(Theta):** The maximum angle between any two UDs in a dependency graph.

2.3 Abstraction

We explain the abstraction of hierarchical dependency graphs with an example. Consider the dependency graph of size 9 x 9, which is similar to the transitive closure design, in Figure 3(a). Consider the time instant at which all the nodes in the graph are busy processing data. At the beginning of each cycle, each node gets inputs from top and left directions, processes data and outputs updated data to the right and bottom directions. Similarly for every cycle, there is input from top and left directions and output to the right and bottom directions flowing for the overall dependency graph. Similar to transitive closure design, there are two UDs for this dependency graph along the $\hat{\imath}$ and $\hat{\jmath}$ directions and the UUD is along $(\hat{\imath}+\hat{\jmath})/\sqrt{2}$.

We have re-drawn Figure 3(a) in Figure 3(b) with the following modifications. Cut the dependency graph along six lines parallel to the UDs, three in each direction as shown as dashed lines. Represent each partition with a shaded

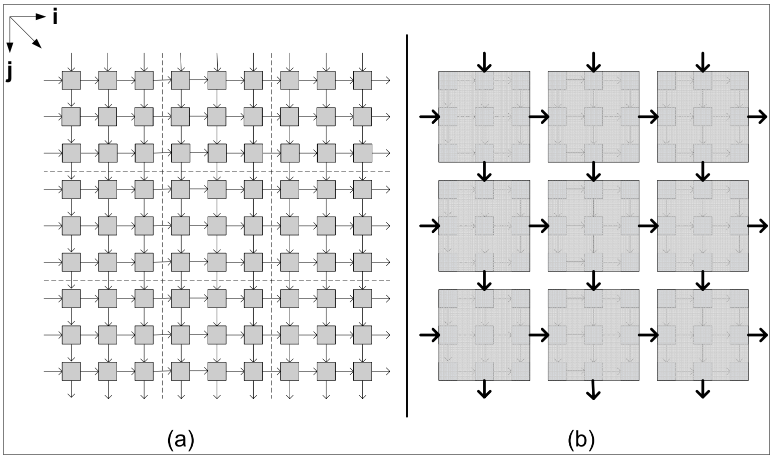


Fig. 3. Hierarchical Systolic Array

big node (name it macro node), encompassing the 3 x 3 matrix of nodes (name them micro nodes) inside it. Represent the data flow into and out of each macro node with a bold arrow. For e.g., for the top left macro node, the arrow from above represents input coming to all the top 3 micro nodes inside it, the arrow from left represents input coming into all 3 micro nodes on the left. Similarly, the right and bottom arrows represent output from all three right and bottom micro nodes respectively, inside it. This is true for all macro nodes in Figure 3(b).

***Formalizing the Abstraction*:** The idea of bisecting a dependency graph into parts and representing each partition by a macro node is called **Hierarchical Dependency Graphs** (HDGs). In this paper, we discuss about HDGs of one level of hierarchy: a **macro dependency graph** consisting of a set of macro nodes, which is nothing but the complete dependency graph being studied from the point of view of the macro nodes. Each macro node has a **micro dependency graph** (one macro node with micro nodes) inside it. While this abstraction some similarity to tiling ([4]), this idea can be more easily extended to multiple levels of hierarchies, for new generation high performance computing and supercomputing systems with various levels of parallelism and compute power organization.

2.4 Mapping Methodology

In this Section, we first explain the mapping technique and provide an argument for its viability. In both these instances, we will be discussing in terms of the number of cycles of operation of macro nodes. So we first characterize this idea.

Characterization of c cycles of a macro node*:** A dependency graph operates in cycles. This means, in each cycle, a node gets input at the beginning of a cycle, processes the data and outputs data at the end of the cycle. We refer to ***c cycles of operation of a macro node as, all micro nodes inside the macro node execute ***c*** cycles of operation each, taking care of the dependencies between the macro node and the remaining part of the macro dependency graph and also, dependency between the micro nodes themselves inside the macro node.

Steps for mapping a Dependency Graph to a Multicore Processor: The following are the steps for generating a mapping for a dependency graph on to a multicore processor:

- Cut the dependency graph along the UDs and represent each partition as a macro node.
- Schedule the macro nodes along the UUD taking care of the dependencies between macro nodes, by assigning ***c*** cycles of operation of a macro node to each core.
- It is possible to schedule in parallel, all macro nodes perpendicular to the UUD. This should be done step by step, along the UUD.

By stating properties of HDGs below, we will show that there is enough parallelism between the macro nodes that many macro nodes can be scheduled in parallel to many cores which can operate independently. Also, the above methodology will become clearer by examining these steps in the case studies in Section 3. There is also one more issue: the value of ***c***. This will also be discussed after the properties of HDGs.

Viability of the mapping technique: We describe properties of the Hierarchical Dependency Graphs to show the viability of our mapping technique.

Property of Parallelism between Macro Nodes: Extending the property of parallelism in Section 2.2, at a single instant of time, several macro blocks can be processing data in parallel.

Basis for the property: We know that the conception of dependency graphs and systolic array designs is to extract the parallelism in the algorithm. At a single instant of time, many nodes in a DG can be processing data in parallel and passing data between each other at the end of each cycle (property of parallelism in Section 2.2). The above property is merely extending that parallelism from micro nodes to the level of macro nodes.

Property of Independent Operation of a Macro node: There exists at least one scheduling order by which, each of the macro blocks can be processed independent of the other for ***c*** cycles of operation of the macro node. The value of ***c*** is discussed below.

Basis for the property: This property is extension of the property of modularity, regularity and scaling in Section 2.2.

Value of c: The value of c is of important concern for us since, it decides the number of cycles for which a macro node can be processed independently by a core. This is directly related to taking care of the dependencies between macro nodes and hence automatically parallelizing the complete dependency graph on to the multicore processor. There can be two variations in the value that c can take based on the Θ of a dependency graph.

– $0^o \leq \Theta \leq 90^o$: We show the two boundary cases in the two case studies, and prove that c can take the total number of cycles of the dependency graph in this scenario. If there is only one UD for the dependency graph, it falls into the category of $\Theta = 0^o$. This being the case of convolution, transitive closure has $\Theta = 90^o$.
– $90^o < \Theta \leq 180^o$: The value c can take varies depending on the DG design and how the dependencies between nodes and macro blocks are arranged. Also, most common DG designs fall into the previous category. In this paper, we do not consider this case and plan to study this in future work.

The two cases $180^o < \Theta \leq 270^o$ and $270^o < \Theta \leq 360^o$ can be interpreted as $90^o < \Theta \leq 180^o$ and $0^o < \Theta \leq 90^o$ respectively.

3 Case Studies

We present case studies of two algorithms on two architectures and we show scalable results in all four cases. First, we present a simple generic model of a multicore processor. We use this model to explain the mapping of the algorithm to a generic multicore processor, hence providing support to our claim that our methodology of mapping dependency graphs is applicable to multicore processors in general. Then we give a brief description of the two multicore platforms and the details of their architectures. In Sections 3.2 and 3.3, we present the mapping of transitive closure and convolution to multicore processors using the generic model as explained above. In Section 3.4, we discuss the experimental results for the two algorithms on the two platforms.

3.1 Architecture Summaries

The multicore processor model is shown in Figure 4. The chip has m cores, each core having a local cache (LC). These local cache access main memory via a memory controller. We give a brief description of the architectures below and explain more details wherever necessary later in the paper (Section 3.4).

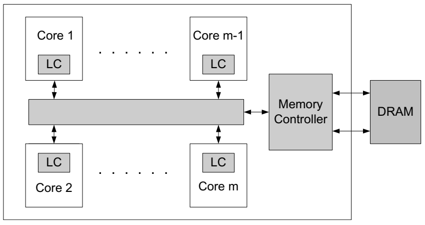


Fig. 4. Generic model of a multicore processor

Intel Quad Core Processor: One of our platforms is a state-of-the-art homogeneous multicore processor system: Intel Xeon quadcore system. It contains two Intel Xeon x86_64 E5335 processors, each having four cores. The processors run at 2.00 GHz with 4 MB cache and 16 GB memory. The operating system is Red Hat Enterprise Linux WS release 4 (Nahant Update 7). We installed GCC version 4.1.2 compiler and Intel C/C++ Compiler (ICC) version 10.0 with streaming SIMD extensions 3 (SSE 3), also known as Prescott New Instructions (PNI).

Cell Broadband Engine: The Cell BE processor ([9]) is one of the first heterogeneous multicore processors that has given the programmer explicit control of memory management and low level communication primitives between the cores on the chip. It consists of a traditional microprocessor (PPE) that controls 8 SIMD co-processing units (SPEs), a high speed memory controller and a high bandwidth bus interface (EIB), all integrated on a single chip. Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC) and 256 KB of local store (LS) where the MFC is a software controlled unit serving the memory management between the LS and main memory. This utility of software controlled LS allows more efficient use of memory bandwidth than is possible with standard prefetch schemes on conventional cache hierarchies, but also poses extra challenges for the programmer. At 3.2 GHz, the single precision peak performance of the Cell processor is 204.8GFLOPS with Fused Multiply Add (two ops) primitive and 102.4GFLOPS without it (single op).

3.2 Case 1: Transitive Closure

Algorithm Description: The algorithm and the dependency graph design for transitive closure are described in Section 2.2. So, we directly describe the mapping. We consider here sample sizes of the problem and the cut sets for ease of illustration. The actual numbers can be varied depending on the real world problem sizes. Examples of these experiments are given in the Section 3.4.

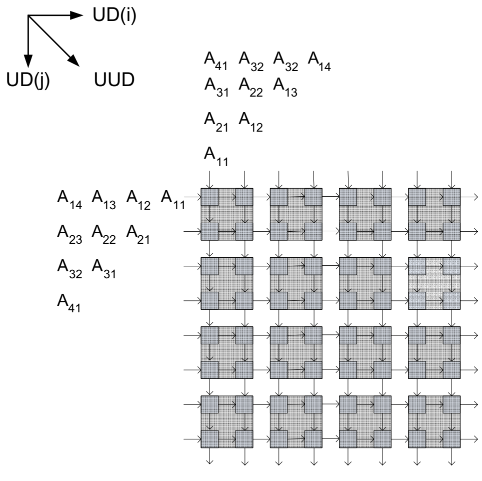


Fig. 5. Hierarchical Dependency Graph for Transitive Closure

Consider transitive closure problem of size $N = 8$. So we have an adjacency matrix of size 8 x 8. Figure 5 shows this 8 x 8 adjacency matrix. As described previously, there are two UDs along $\hat{i}$ and $\hat{j}$ and the UUD is along $(\hat{i}+\hat{j})/\sqrt{2}$. Also, as previously explained, we refer to one cycle of operation of a node as, the node taking input from top and left directions, processes it and send output to the left and bottom directions at the end of the cycle. The total number of cycles of operation of each node in the graph is equal to $N = 8$.

Mapping: We follow the steps of mapping as described in Section 2.4. First, cut the dependency graph with 8 cut lines, 4 in each direction parallel to the UDs. Figure 5 shows the 2 x 2 macro blocks shaded, after cutting the graph. Observe that this Figure is very similar to Figure 3(b) where the difference is that the macro blocks are of size 3 x 3, and the problem size is $N = 9$ in that Figure. Next, schedule the macro nodes along the UUD and execute each macro node for "N cycles of the macro node" (Section 2.4). Therefore, $\boldsymbol{c}$ is equal to the complete cycles of the macro node. Also, schedule macro nodes perpendicular to the UUD in parallel to multiple cores, which process those macro nodes independently.

Numbering the top left macro node and micro node (0, 0) and $\hat{i}$ and $\hat{j}$ axis in Figure 5, the above steps of mapping have the following interpretation:

1. A single core processes macro node (0, 0) (here processes means update the macro node for all N cycles, i.e., micro node (0, 0) is processed for 8 cycles, then micro nodes (1, 0) and (0, 1) are processed for 8 cycles, then micro node (1, 1) is processed for 8 cycles).
2. Process macro nodes (0, 1) and (1, 0) in parallel by two cores independently.
3. Process macro nodes (0, 2), (1, 1) and (2, 0) in parallel by three cores independently.
4. Process macro nodes (0, 3), (1, 2), (2, 1) and (3, 0) in parallel by four cores independently.
5. Process macro nodes (1, 3), (2, 2) and (3, 1) in parallel by three cores independently.
6. and so on ...

There is a synchronization point after each step, which can be removed once the number of macro nodes processed in parallel in a step exceeds the number of cores, m on the chip.

The above mapping and scheduling takes care of the dependencies between all nodes of the dependency graph in Figure 2. Also, observe that the HDG satisfies the properties described in Section 2.2. The first property states that there will be parallelism between the macro nodes. This is true since, with large problem sizes, the number of macro blocks that can be scheduled in parallel increases, and all m cores will be busy. The second property states that, there exists a scheduling order by which, the macro blocks can be processed independent of each other for $\boldsymbol{c}(= N)$ cycles of operation of the macro node. This is also true since, all the macro blocks along the perpendicular to the UUD have no dependencies and can be processed for $\boldsymbol{c}(= N)$ cycles of operation. Also by respecting the dependencies between macro blocks along the UUD by a synchronization point wherever necessary, the macro blocks along the UUD are also scheduled for N cycles of operation.

3.3 Case 2: Convolution

Algorithm Description: Convolution is one of the most important kernels in scientific computing. It is of fundamental importance [11] in signal processing, image processing, communication systems, computer vision and pattern recognition algorithms. Variations of convolution [5] are also used to solve integer multiplication and polynomial multiplication problems. There has been a lot of interest for high performance convolution computation recently [7], [8]. We describe the mapping of 1D convolution dependency graph to a multicore processor using our technique. A good feature of the 1D dependency graph design is that, 2D and 3D designs are exact symmetric and modular extensions of the 1D design. So our mapping technique is directly applicable to 2D and 3D convolution also.

Consider two digital signals A and B of dimension 1 x N. The convolution C of A and B, represented by C = A $\otimes$ B, is a 1 x $2N - 1$ given by equation 3,

$$C(i) = \Sigma A(i).B(N - i) \tag{3}$$

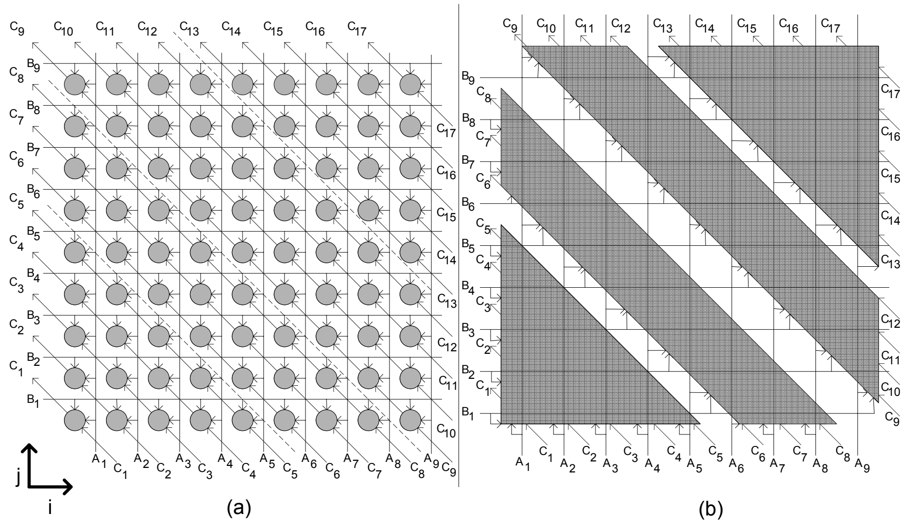


Fig. 6. Dependency Graph (DG) and Hierarchical DG Abstraction for Convolution

Consider a problem of size $N = 9$. The dependency graph for computing the convolution of A and B is shown in Figure 6(a). Unlike transitive closure where all data flowing in the dependency graph is being updated, here not all data is updated. So, as described in Section 2.2, we differentiate the two types of data flowing. So in the Figure, we show the data that is not being updated (signals A and B) with black coloured lines and data being updated (signal C) with blue coloured lines. With $\hat{i}$ and $\hat{j}$ axis as shown, there is a single UD along $(\hat{-i}+\hat{j})/\sqrt{2}$. Since there is only one UD, that itself will become the UUD.

Mapping: Using the steps of mapping as described in Section 2.4, cut the dependency graph along the UD. Figure 6(b) shows the dependency graph after this operation where the four partitions resulting from three cut lines (shown in dashed lines in Figure 6(a)): after C_5, C_8 and C_{12}, are shown in shaded blocks. Each partition, a macro block, is represented by a shaded region with its inputs and outputs. A single line can be drawn perpendicular to the UUD which will pass through all macro nodes, which means there is no dependency between them and all the four macro nodes can be scheduled in parallel to four cores independently. Hence, when there are m cores, the dependency graph is partitioned into m macro blocks, with equal computational load, each of which will be processed by each core independently. Similar to the transitive closure case, observe that the HDG for convolution satisfies the two properties in Section 2.2.

3.4 Experimental Results and Discussion

We provide experimental results below proving that the mapping techniques give scalable results on multicore processors. We comment about the performance (single precision) and optimizations in Section 3.4.

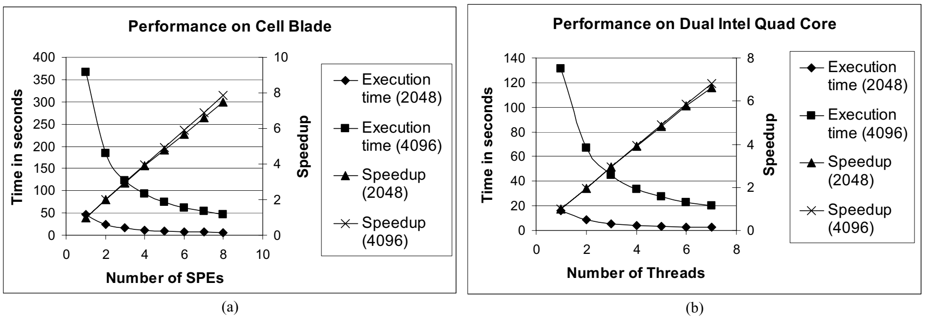
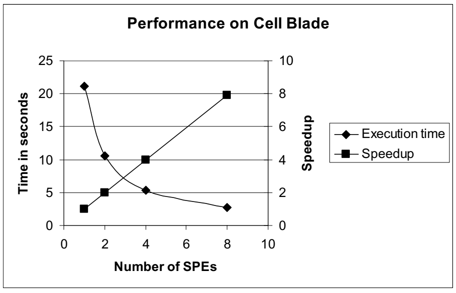


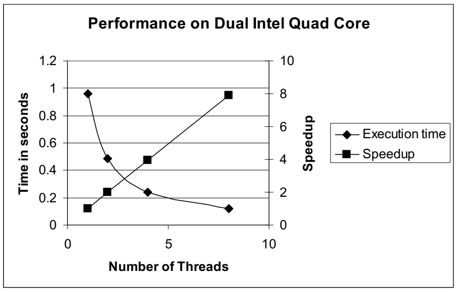
Fig. 7. Experimental results for Transitive Closure

Transitive Closure: We considered problems of sizes $N = 2048$ and 4096. Each macro block is of size 64 x 64 by using 32 and 64 cut lines in each direction parallel to the UDs respectively for the two problem sizes. Hence, there are a total of 32 x 32 and 64 x 64 macro blocks for problem sizes 2048 and 4096 respectively. Figure 7(a) shows the scalable results on Cell blade. The performance achieved is 4.42GFLOPS. Figure 7(b) shows the scalable results on the dual Intel quad core platform. We achieved 10.54GFLOPS on this platform.

Convolution: We considered problem of size $N = 32$K. Figure 8(a) shows performance results on Cell Blade. We achieved 0.801GFLOPS on this platform. Figure 8(b) shows the results on the Intel quad core platform. The performance achieved is 17.5GFLOPS. Both the results are scalable as can be observed from the Speedup plots.



(a)



(b)

Fig. 8. Experimental results for Convolution

Discussion: Before commenting about the performance itself, we need to describe the differences and factors in architecture and programming models for the Cell and Intel multicore processors that effect the performance. First, in the Cell processor, as mentioned in Section 3.1, the complete memory management between the main memory and local stores of SPEs is in the hands of the programmer. While this increases the complexity of programming, it gives an advantage for achieving performance in cases where the operating system and cache coherency protocols cause heavy overhead and limit the performance. Whereas, in the Intel quad core, memory management is completely handled by the operating system and compiler. This difference also allows us to do a baseline implementation comparison (Figure 10) with our techniques on the Intel processor but not on the Cell (here baseline implementation means the algorithm implementation without our parallelization techniques and only with compiler optimizations). This is because, unlike the Intel processor where the compiler can execute a piece of code, on the Cell processor, we need to parallelize and take care of memory management right for the most basic implementation.

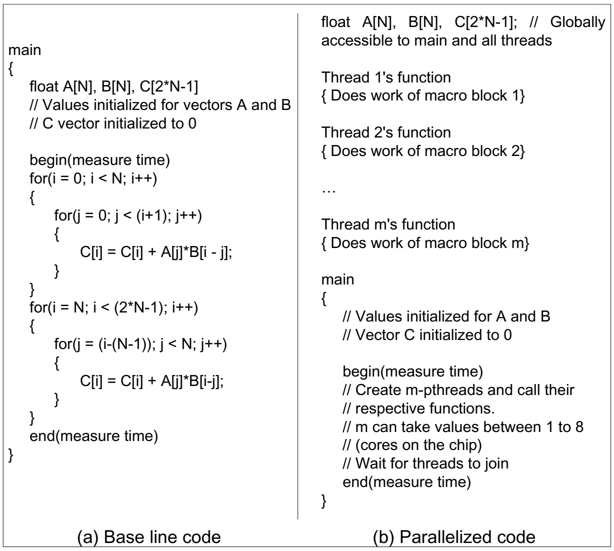
Second, the SPEs in the Cell processor are highly specialized for SIMD processing units but with pre-condition that the data should be aligned in memory. SIMDization is not possible with mis-aligned data and which means operations on mis-aligned data ([9]) should be loaded in a single preferred slot of a vector register, the data is processed and written back to memory from the preferred slot. This causes a heavy overhead decreasing the performance by orders of magnitude ([9]). SIMDization is more easy on the Intel quad core, in fact the Intel ICC compiler is good enough to auto-simdize operations quite efficiently.

Lastly, Cell SPEs have a bad branch prediction units compared to the Intel's cores. This is because SPE's architecture is optimized for streaming data ([9]).

Performance of Transitive Closure: We achieved a high performance of 10.54GFLOPS on the Intel Quad core processor. The compiler auto-simdizied the code. We manually SIMDized the code on the Cell processor by grouping operations on micro node into a vector operation. But at the end of N cycles, one value (the lowest) out of the four should be assigned to the micro node. There is no vector primitive that does this on the Cell leading to a non-SIMD operation and also a branch prediction operation. This lead to the drop in performance and we achieved 4.42GFLOPS. With an improved branch prediction unit and more hardware to take care of non-SIMD operations in future versions of Cell ([9]), there is a very promising possibility to achieve peak performance of the algorithm on the Cell.

Performance of Convolution: We achieved 17.5GFLOPS on the Intel platform, whereas we achieved only 0.801GFLOPS on the Cell. This is because, the convolution algorithm necessitates operations on non-aligned data in every clock cycle. As these operations are not SIMDized, they lead to heavy loss in performance on the Cell processor. As mentioned above, this is one aspect that has to be addressed in future versions of the Cell processor to improve its potential to wide range of algorithms. Also, our performance of 17.5GFLOPS on the Intel platform is higher than other recent research work for high performance computation of convolution. In different contexts of the same convolution operation, researchers have achieved 2.16GFLOPS per node ([7]) on a 4-rack Blue Gene system (4096 nodes leading to overall 7TFLOPS) and 3.16GFLOPS ([8]) on an FPGA platform.

Comparing with Compiler Optimizations: To measure the impact of our parallelization with the compiler optimizations, we ran a baseline implementation for convolution which involved straight coding of the algorithm with compiler optimizations (-O2, -O3, -O4, -msse3 etc.) both with GCC and Intel's ICC compiler. The pseudo codes for the baseline implementation and our parallelized version are shown in Figure 9. Figure 10 shows the results from this experiment. The best possible performance achieved by a compiler is 2.2GFLOPS (Baseline for ICC) whereas with our parallelization, we achieved 17.5GFLOPS. This shows that our mapping technique has given the compiler more opportunities to parallelize and optimize the dependency graph computations on the multicore processor.



```
main
{
    float A[N], B[N], C[2*N-1]
    // Values initialized for vectors A and B
    // C vector initialized to 0

    begin(measure time)
    for(i = 0; i < N; i++)
    {
        for(j = 0; j < (i+1); j++)
        {
            C[i] = C[i] + A[j]*B[i - j];
        }
    }
    for(i = N; i < (2*N-1); i++)
    {
        for(j = (i-(N-1)); j < N; j++)
        {
            C[i] = C[i] + A[j]*B[i-j];
        }
    }
    end(measure time)
}
```

(a) Base line code

```
float A[N], B[N], C[2*N-1]; // Globally
accessible to main and all threads

Thread 1's function
{ Does work of macro block 1}

Thread 2's function
{ Does work of macro block 2}

...

Thread m's function
{ Does work of macro block m}

main
{
    // Values initialized for A and B
    // Vector C initialized to 0

    begin(measure time)
    // Create m-pthreads and call their
    // respective functions.
    // m can take values between 1 to 8
    // (cores on the chip)
    // Wait for threads to join
    end(measure time)
}
```

(b) Parallelized code

Fig. 9. Pseudo code for convolution

4 Conclusion

We summarize the contributions made by this paper. Starting from a seminal problem of mapping systolic arrays to multicore processors, we made the observation that mapping of dependency graphs is more fundamental and should be studied rather than systolic arrays. We defined the abstraction of Hierarchical Dependency Graphs, using which we proposed a mapping methodology to map and parallelize a dependency graph to a multicore processor. We presented two case studies and achieved scalable results and good performance illustrating our methodology. In future, we plan to conduct more case studies of mapping dependency graphs to multicore processors and also possibly integrate these techniques into compilers.

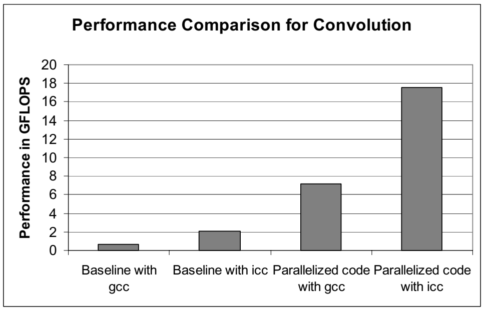


Fig. 10. Comparison with Compiler Optimizations

References

1. Kung, S.Y.: VLSI Array Processors. In: Kailath, T. (ed.) Prentice-Hall, Englewood Cliffs (1988)
2. Ullman, J.D.: Computational aspects of VLSI. Computer Science Press (1983)
3. Kung, H.T., Leiserson, C.E.: Systolic arrays (for VLSI). In: Sparse Matrix Symposium, pp. 256–282. SIAM, Philadelphia (1978)
4. Penner, M., Prasanna, V.K.: Cache Friendly Implementations of Transitive Closure. In: Proc. of PACT (2001)
5. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes. Morgan Kaufmann, San Francisco (1992)
6. Rao, S.K., Kailath, T.: Regular Iterative Algorithms and their Implementation on Processor Arrays. Proc. of the IEEE 76, 259–269 (1988)
7. Nukada, A., Hourai, Y., Nishada, A., Akiyama, Y.: High Performance 3D Convolution for Protein Docking on IBM Blue Gene. In: Stojmenovic, I., Thulasiram, R.K., Yang, L.T., Jia, W., Guo, M., de Mello, R.F. (eds.) ISPA 2007. LNCS, vol. 4742, pp. 958–969. Springer, Heidelberg (2007)
8. Huitzil, C.T., Estrada, M.A.: Real-time image processing with a compact FPGA-based systolic architecture. Journal of Real-Time Imaging (10) 177–187 (2004)
9. Arevalo, A., Matinate, R.M., Pandlan, M., Peri, E., Ruby, K., Thomas, F., Almond, C.: Prog. the Cell Broadband Engine: Examples and Best Practises, IBM Redbooks
10. Karp, R.M., Miller, R.E., Winograd, S.: The Organization of Computations for Uniform Recurrence Equations. Jour. of ACM 14(3), 563–590 (1967)
11. Oppenheim, A.V., Schafer, R.W., Buck, J.R.: Discrete-Time Signal Processing. Prentice Hall Signal Processing Series (2004)

A Tool for Detecting First Races in OpenMP Programs[*]

Mun-Hye Kang[1], Ok-Kyoon Ha[1], Sang-Woo Jun[2], and Yong-Kee Jun[1,**]

[1] Gyeongsang National University, Jinju, 660-701 South Korea
[2] Seoul National University, Seoul, 151-742 South Korea
kturtle@hanmail.net,jassmin@gnu.ac.kr,aradia.jun@gmail.com,jun@gnu.ac.kr

Abstract. First race detection is especially important for effective debugging, because the removal of such races may make other affected races disappear. The previous tools can not guarantee that detected races are the first races to occur. We present a new tool to detect first races in a program with nested parallelism using a two-pass on-the-fly technique. To show accuracy, we empirically compare our tool with previous tools using a set of synthetic programs with OpenMP directives.

Keywords: OpenMP programs, first races to occur, race detection.

1 Introduction

A data race [1,6] or simply a race occurs when there are two conflicting accesses from different threads to a shared variable without appropriate synchronization, and at least one access is a write. Detecting these races is important for debugging programs with OpenMP directives, because races result in unintended non-deterministic executions of the program. For effective debugging, it is especially important to detect the first data races to occur (first races), because the removal of such races may make other affected races disappear or appear. The previous tools for debugging the races in OpenMP [7] programs include Intel Thread Checker [2,9] and Sun Thread Analyzer [8,9]. But, these tools can not guarantee that detected races are the first races.

This paper presents an effective tool that reports the first races in programs with nested parallelism using a labeling scheme called NR Labeling [5] and a race detection protocol [4]. The NR generates logical concurrency information for every thread in an execution, and does not require central bottlenecks that using local data structures require. The protocol detects the races by monitoring candidate accesses that may be involved in first races during two passes of

[*] This research was supported by the MKE (The Ministry of Knowledge Economy), Korea,under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute for Information Technology Advancement) (IITA-2009-C1090-0904-0001).

[**] Corresponding author: In Gyeongsang National University, he is also involved in the Research Institute of Computer and Information Communication (RICIC).

program executions. This paper presents our tool which detects the first races during the two passes of program execution, and empirically shows with a set of synthetic programs that our tool is practical with respect to accuracy.

2 Background

A data race [1,6] or simply a race occurs when there are two conflicting accesses from different threads to a shared variable without appropriate synchronization, and at least one access is a write. Since such races result in unintended non-deterministic executions of programs, it is important to detect the races for the effective debugging of such programs. To help user's understanding, an execution of a parallel program is represented by a directed acyclic graph called POEG (Partial Order Execution Graph) [3] as shown in Fig. 1. In a POEG, a vertex indicates a fork or join operation, and an arc between vertices represents a forked or joined thread. The accesses named r and w drawn in the figure as small dots on the arcs represent a read and a write access, that access the same shared variable, respectively. The numbers in the access names indicate the order in which those accesses are observed.

An access e_j is affected by another access e_i, if e_i happened before e_j and e_i is involved in a race. A race e_i-e_j is unaffected, if neither e_i nor e_j are affected by any other accesses. A race is partially affected, if only one of e_i and e_j is affected by another access. A tangle is a set of partially affected races such that if e_i-e_j is a race in the tangle then exactly one of them is affected by e_k such that e_k is also involved in a race of the same tangle. A tangled race is a partially affected race that is involved in a tangle. A first race [4] to occur (first race) is either an unaffected race or a tangled race. Fig. 1 shows a POEG which includes eleven races, but only two of the eleven races can be the first races. Eliminating the two possibly first races is very important for effective debugging, because it may make the other nine affected races disappear or appear.

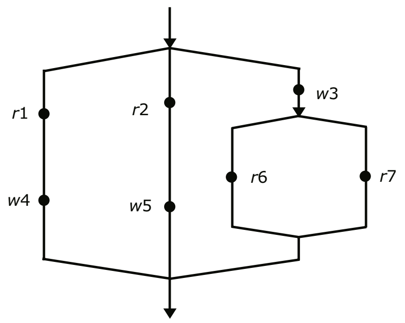


Fig. 1. The POEG

	Synthetic			detected races		
				Our Tool	Checker	Aanlyzer
1	r1 w3		r2 w4	r1-w4 r2-w3	r1-w4 r2-w3	-
2	r1 r4	r2 r5	w3 r6	r1-w3 r2-w3	-	r2-w3 r1-w3
3	r1	r2	r3	r2-w4	w4-r2	-
4	r1 w4	r2 r5	w3 r6	r1-w3 r2-w3	-	-
5	r1 w4	r2 r5	w3 r6 r7	r1-w3 r2-w3	-	-

Fig. 2. The Results for Accuracy

The previous tools for debugging races in OpenMP programs include Intel Thread Checker [2,9] and Sun Thread Analyzer [8,9]. Thread Checker sequentially executes a program to project parallel threads by instrumenting the source code with additional codes, and then detects races by checking data dependency during an execution of the program. The tool does not verify the existence of races, including the first racesin an execution. In Fig. 1, Thread Checker detects six races {$w4$-$r2$, $w4$-$w5$, $r1$-$w5$, $w5$-$w3$, $w5$-$r6$, $w5$-$r7$}, but these races are affected by two first races. Thread Analyzer has not been published the internal mechanism of Thread Analyzer, and it has not been analyzed with respect to its functionality for detecting races in OpenMP programs. Unfortunately, by experimenting it with just one small set of synthetic programs we discovered that the tool can not guarantee that detected races are the first races. In Fig. 1, Thread Analyzer detects seven races, but these races are also affected by first races.

3 The First Race Detection Tool

Fig. 3 shows the client-server structure of our tool. In the client side, there are four modules: selector, instrumentor, sender, and reporter. In the server side, there is the compiler, and run-time libraries for race detection. First, when the source codes to be debugged are admitted, the selector module helps the users select shared variables and the information of race detection protocol to be used for monitoring. The scanner module analyzes the source program to add the protocol libraries, and the instrumentor module adds the race detection protocol into the source code. The instrumented source program is transferred to the server by the sender module in the client. The server executes the instrumented object program compiled and linked with the library of protocol engine and transfers the results of race detection to the client. Finally, the reporter module in the client notifies the results to the user.

For detecting the first races, the protocol [4] implemented in run-time libraries in the server efficiently detects the races in programs with nested parallelism during two passes of program executions. In the first pass, the protocol collects candidate accesses that may be involved in first races, keeping a constant number

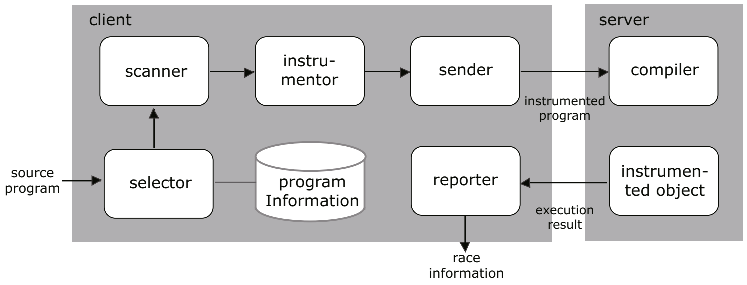
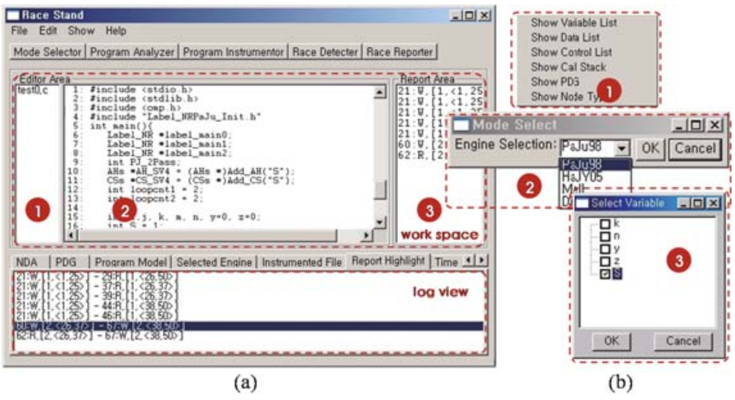


Fig. 3. Design of the first race detection tool



```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "Label_NRPaJu_Init.h"
int main(){
    Label_NR *label_main0;
    Label_NR *label_main1;
    Label_NR *label_main2;
    int PJ_2Pass;
    AHs *AH_SV4 = (AHs *)Add_AH("S");
    CSs *CS_SV4 = (CSs *)Add_CS("S");
    int loopcnt1 = 2;
    int loopcnt2 = 2;

    int i,j, k, m, n, y=0, z=0;
    int S = 1;
```

```
21:W,[1,<1,25>] - 29:R,[1,<26,50>]
21:W,[1,<1,25>] - 37:R,[1,<26,37>]
21:W,[1,<1,25>] - 39:R,[1,<26,37>]
21:W,[1,<1,25>] - 44:R,[1,<38,50>]
21:W,[1,<1,25>] - 46:R,[1,<38,50>]
60:W,[2,<26,37>] - 67:W,[2,<38,50>]
62:R,[2,<26,37>] - 67:W,[2,<38,50>]
```

Fig. 4. Interface of Effective tool

of accesses stored for each shared variable during an execution. In the second pass, each candidate collected in the first pass is also examined based on the *happened-before relation* and the *left-of relation* if the current access is conflicting with the candidate access, in order to complete the set of candidate accesses collected in the first pass. This protocol is efficient with regard to execution time and memory space, because a small constant is the number of accesses stored for each shared variable during the execution. The protocol detects all but one of the first race in the worst case.

Fig. 4 shows the programm interface which consists of four components: a classical menu bar, a work space, a tool bar, and a log view. The work space consists of three parts: a list of opened program files, a source code window to display the selected program, and the information of detected races. The tool bar consists of four buttons: Mode Selector, Program Analyzer, Program Instrumentor, and Race Detector. These buttons are arranged to reflect the procedure of detecting races. The Mode Selector invokes a window as shown in Fig. 4(b) to select the protocol. The Program Analyzer extracts information from the source program to detect the first races in programs with nested parallelism, and then shows a window for selecting a shared variable to be monitored. And, the log view shows the results of Program Analyzer. The Program Instrumentor adds the run-time library of a race detection protocol in the source code. The Race Detector creates an executable file by compiling the instrumented code which calls the run-time library of the protocol in the server and the Reporter presents information on detected races. Each entry of the information includes a line number and two accesses which are represented with event types and thread labels. If the user clicks twice highlighted lines of the log view using the Report Highlight tab, the tool highlights every line of the corresponding source code.

The client programs of the tool is implemented in the Java language and developed in Eclipse environment using Java Development Kit version 1.6. The run-time libraries of the tool in the server are implemented in the C language.

We installed Intel C/C++ version 10 to compile OpenMP programs and developed a socket program to transfer files. To experiment with previous tools, we installed Thread Checker 3.1 for Linux and Thread Analyzer of Sun Studio 12. To show accuracy of our tool, we empirically compare the tool with the previous tools using a set of synthetic programs with OpenMP directives. Synthetic programs were developed by varying the thread number, the nesting depth, and the location or number of write events. Fig. 2 shows the test results for accuracy from five types of synthetic programs. The first program shows two parallel threads using two columns and the next three types of programs have three parallel threads. The remaining type of programs start three parallel threads of which the last thread forks two child threads. In the result, our tool detected only the first races to occur in every kind of synthetic programs, and Thread Checker and Analyzer did not detect the first races in the many types of programs.

4 Conclusion

This paper presents a novel tool that detects the first races in the program with nested parallelism using a two-pass on-the-fly technique. We empirically compare our tool with the previous tools using a set of synthetic programs with OpenMP directives, and the results support that our tool efficiently detects only the first races in all kinds of synthetic programs.

References

1. Banerjee, U., Bliss, B., Ma, Z., Petersen, P.: A Theory of Data Race Detection. In: Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD), Portland, USA, pp. 69–78. ACM, New York (2006)
2. Banerjee, U., Bliss, B., Ma, Z., Petersen, P.: Unraveling Data Race Detection in the Intel Thread Checker. In: Workshop on Software Tools for Multi-core Systems (STMCS), Portland, USA, pp. 69–78. ACM, New York (2006)
3. Dinning, A., Schonberg, E.: An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In: 2nd Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 1–10. ACM, New York (1990)
4. Ha, K., Jun, Y., Yoo, K.: Efficient On-the-fly Detection of First Races in Nested Parallel Programs. In: Workshop on State-of-the-Art in Scientific Computing (PARA), Copenhagen, Denmark, June 2004, pp. 75–84 (2004)
5. Jun, Y., Koh, K.: On-the-fly Detection of Access Anomalies in Nested Parallel Loops. In: 3rd ACM/ONR Workshop on Parallel and Distributed Debugging (WPDD), San Diego, California, pp. 107–117. ACM, New York (1993); also in SIGPLAN Notices 28(12), 107–117 (1993)
6. Netzer, R.H.B., Miller, B.P.: What Are Race Conditions? Some Issues and Formalizations. Letters on Prog. Lang. and Syst. 1(1), 74–88 (1992)
7. OpenMP Architecture Review Board, OpenMP Application Programs Interface, Version 3.0 (May 2007)
8. Sun Microsystems, Inc.: Sun Studio 12: Thread Analyzer User's Guide (2007)
9. Terboven, C.: Comparing Intel Thread Checker and Sun Thread Analyzer. In: Minisymp. on Scalability and Usability of HPC Prog. Tools, In PARCO (September 2007)

Load Balancing of Parallel Block Overlapped Incomplete Cholesky Preconditioning

Igor Kaporin and Igor Konshin

Dorodnicyn Computing Center of Russian Academy of Sciences,
Vavilov str. 40, 119333 Moscow, Russia
{kaporin,konshin}@ccas.ru

Abstract. A modification of the second order Incomplete Cholesky (IC) factorization with controllable amount of fill-in is described and analyzed. This algorithm is applied to the construction of well balanced coarse-grain parallel preconditioning for the Conjugate Gradient (CG) iterative solution of linear systems with symmetric positive definite matrix. The efficiency of the resulting parallel algorithm is illustrated by a series of numerical experiments using large-scale ill-conditioned test matrices taken from the collection of the University of Florida.

Keywords: symmetric positive definite matrix, incomplete Cholesky factorization, conjugate gradient method, parallel preconditioning.

1 Introduction

An analysis of the parallel performance of parallel preconditioned CG solver described, e.g., in [1], [3], showed that the imbalance of the independent tasks at the stage of preconditioning application (due to the loosely controlled variations in the density of the IC factors of submatrices) is mainly responsible for the loss in the overall parallel efficiency.

A simple strategy considered below is to drop down the smallest entries in the most filled incomplete factors. Both matrix theory and numerical experiments show that a slight deterioration in the preconditioning quality should be more than compensated by the reduction of the time cost per iteration.

For the second order IC factorization [2] we propose to reassign the largest by magnitude entries from the error matrix to the IC factor. It may improve both the balancing and convergence rate.

2 Theoretical Analysis of the Post-filtering Techniques

Let us consider separately the cases of the (plain) IC factorization and the IC2 factorization [2] (with the structured error term). Hereafter, we will assume that A is symmetric positive definite matrix symmetrically scaled to the unit main diagonal.

2.1 Plain IC Truncation

Let the IC factor U be obtained from the standard IC equation

$$A = U^T U - S \; , \tag{1}$$

where S is the error matrix, each element of which is $O(\tau)$ and τ is the truncation parameter, $0 < \tau \ll 1$. When one applies a Jennings–Malik type algorithm [5], the matrix S is symmetric nonnegative definite and therefore

$$\lambda(U^{-T} A U^{-1}) \leq 1 \; ,$$

i.e., the eigenvalues of the preconditioned matrix is bounded by 1. At the same time, most eigenvalues of the preconditioned matrix are clustered around 1, which improves the numerical stability of the the corresponding preconditioned CG iterations [7].

Now we consider the splitting of U into the 'main' term and the 'error' term:

$$U = \tilde{U} + \tilde{R} \; ,$$

where the entries of strictly upper triangular matrix R also do not exceed τ in magnitude.

Using the IC equation (1), one has then

$$\begin{aligned} A &= (\tilde{U} + \tilde{R})^T (\tilde{U} + \tilde{R}) - S \\ &= \tilde{U}^T \tilde{U} + (\tilde{U}^T \tilde{R} + \tilde{R}^T \tilde{U} + \tilde{R}^T \tilde{R} - S) \; . \end{aligned}$$

Hence,

$$\tilde{U}^T \tilde{U} = A + \tilde{S} \; ,$$

where

$$\tilde{S} = S - \tilde{U}^T \tilde{R} - \tilde{R}^T \tilde{U} + \tilde{R}^T \tilde{R} \; ,$$

i.e. $\tilde{U}$ is the exact IC of $A + \tilde{S}$, where the matrix $\tilde{S}$ is the perturbed error matrix. The latter also has $O(\tau)$ entries, but, in general, it is symmetric indefinite. Therefore, the norm $\|\tilde{U}^{-1}\|$ can be unbounded, and the corresponding preconditioning is not robust. Of course, the matrix S can be made positive definite by performing an IC factorization of the modified matrix, e.g., $A + \sigma I_n$ (cf. [6]), but the shift parameter should be as rough as $O(\tau)$ in order to guarantee the robustness. Therefore, the quality of such preconditioner is often insufficient, especially for ill-conditioned matrices A.

2.2 Truncation of IC2 Factor *U*

A special type *structured* error matrix arises in the IC2 factorization [2]

$$A = U^T U + U^T R + R^T U - S \; . \tag{2}$$

where R is a strictly upper triangular error matrix, each element of which is $O(\tau)$, S is a symmetric error matrix, each element of which is $O(\tau^2)$ and τ is the truncation parameter, $0 < \tau \ll 1$.

The following two facts (see [2]) are of key importance:

(a) for the same τ, both IC and IC2 factorizations have quite similar upper bounds on the fill-in for U, and

(b) on the contrary, the condition number of matrix A preconditioned by IC and IC2 are $O(\tau \mathrm{cond} A)$ and $O(\tau(\mathrm{cond} A)^{1/2})$, respectively. This explains the superior performance of IC2 preconditioning observed in its practical use (see, e.g., [2], [3]).

The following proposition can be readily deduced from results of [2].

Theorem 1. *For the preconditioned matrix*

$$M = U^{-T} A U^{-1}$$

defined according to (2), one has the estimate

$$\lambda(M) \le 1 + \gamma \tag{3}$$

whenever

$$R^T R \le \gamma S \tag{4}$$

holds.

Remark 1. A simple method which guarantees the validity of (4) is the use of an **a priori** diagonal shift of the order $O(\tau^2)$ for the original matrix A, cf. [2].

We now consider the splitting

$$U = \tilde{U} + \hat{R}\,, \tag{5}$$

where R includes certain amount of the smallest by magnitude nonzero elements of U, and estimate the quality of the preconditioning obtained by the use of truncated (or *post-filtered*) IC factor $\tilde{U}$.

Theorem 2. *For the preconditioned matrix*

$$\tilde{M} = \tilde{U}^{-T} A \tilde{U}^{-1}$$

defined according to (2) and (5), one has the estimate

$$\lambda(\tilde{M}) \le \frac{1+\gamma}{1 - \hat{\gamma} - 2\sqrt{\gamma\hat{\gamma}}} \tag{6}$$

whenever conditions (4) and

$$\hat{R}^T \hat{R} \le \hat{\gamma} S \tag{7}$$

with $\hat{\gamma} < (\sqrt{\gamma} + \sqrt{1+\gamma})^{-2}$ hold.

Remark 2. The other theoretical properties (e.g., the lower spectral bound) of the post-truncated IC2 remain essentially the same as for the original version presented in [2].

2.3 Truncation of IC2 Error Matrix R

Next we consider the splitting of the IC2 error matrix

$$R = \hat{R} + \tilde{R}\,, \tag{8}$$

where $\hat{R}$ includes certain amount of the *largest* by magnitude nonzero elements of R, and estimate the quality of the preconditioning obtained by the use of *augmented* IC2 factor

$$\tilde{U} = U + \hat{R}\,. \tag{9}$$

It appears that the only difference with the above case of IC2 factor truncation (see Theorem 2) is that the equation $\tilde{R} = R + \hat{R}$ is replaced by $\tilde{R} = R - \hat{R}$. Therefore, it can be readily shown that the corresponding estimate (6) holds under the same conditions (4) and (7).

Hence, in both cases one should take care about keeping the norm of the matrix $\hat{R}$ sufficiently small. However, when the truncation of error matrix R is performed, the norm $\|\hat{R}\|$ is not large since $\|R\|$ is bounded.

On the other hand, it makes sense to include into $\hat{R}$ the largest by the magnitude elements of R, thus making $\tilde{R}$ as small as possible. This well agrees with the following result related to the estimation of the K-condition number

$$\mathrm{K}(M) = \left(\frac{1}{n}\mathrm{trace}(M)\right)^n \Big/ \det M\ ,$$

where M is the preconditioned matrix. (A detailed discussion of the relation of $\mathrm{K}(M)$ to convergence of the CG method can be found in [1], [2].)

Theorem 3. *For the preconditioned matrix*

$$\tilde{M} = \tilde{U}^{-T} A \tilde{U}^{-1}$$

defined according to (2) and (9), the following upper bound for the K*-condition number is valid,*

$$\mathrm{K}(\tilde{M}) \le (\det U)^2 / \det A\ , \tag{10}$$

whenever the condition

$$\tilde{R}^T \tilde{R} \le S + R^T R \tag{11}$$

holds.

Hence, if the matrix $\tilde{R}$ is sufficiently small by the norm, then the K-condition number of the preconditioned matrix has an upper bound which does not depend on this matrix.

3 Finding a Prescribed Amount of Smallest Elements in Array

According to (7), one should include in $\hat{R}$ the smallest entries of U in order to keep $\hat{\gamma}$ as small as possible in the case of post-filtering the IC2 factor. On the contrary, if the IC2 error matrix is filtered, one may choose to find the largest entries of R.

An obvious approach is to sort the whole array U and include into the matrix $\hat{R}$ its smallest entries, which would cost $O(\mathrm{nz}(U)\log \mathrm{nz}(U))$ operations.

To reduce the cost of truncation we use hashing over the interval $[0,1]$ divided into n equal segments, where n is the dimension of the matrix A. After rehashing with the corrected interval boundaries we obtain the required threshold up to 1 element accuracy. The costs of such procedure is about $2n + 2\mathrm{nz}(U)$ operations.

4 A Description of Parallel IC2-Based Preconditioning

Let A be reordered and split in the same way as for the Block Jacobi preconditioning, i.e. the t-th diagonal block of the symmetrically reordered matrix has the dimension n_t and $n_1 + \ldots + n_p = n$. Here $t = 1, ..., p$, and p is the block dimension of A. For the t-th diagonal block, let us define the 'basic' index set as

$$\{k_{t-1}+1, \ldots, k_t\} \ ,$$

where

$$k_{t-1} = n_1 + ... + n_{t-1} \ , \qquad k_0 = 0 \ , \quad k_p = n \ ,$$

and introduce the 'overlapping' index sets as

$$\{j_t(1), \ldots, j_t(m_t - n_t)\} \ , \qquad j_t(p) \le k_{t-1} \ ,$$

where

$$m_t \ge n_t \ , \qquad m_1 = n_1 \ .$$

For each t, the latter index set typically includes those indices not greater than k_t that are the most 'essentially' connected to the basic index set, e.g. in the sense of the sparse matrix graph adjacency relations. According to [1], [3], the Block Incomplete Inverse Cholesky (BIIC) preconditioner H is:

$$H = \sum_{t=1}^{p} V_t U_t^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{n_t} \end{bmatrix} U_t^{-T} V_t^T \ ,$$

where V_t are rectangular matrices composed of unit n-vectors e_j as follows:

$$V_t = [e_{j_t(1)}|...|e_{j_t(m_t-n_t)}|e_{k_{t-1}+1}|...|e_{k_t}] \ , \quad t = 1, ..., p \ ,$$

and each upper triangular matrix U_t is the (approximate) right Cholesky factor of the t-th 'extended' diagonal $m_t \times m_t$ submatrix $V_t^T A V_t$, that is,

$$V_t^T A V_t \approx U_t^T U_t \ , \quad t = 1, ..., p \ .$$

4.1 Block Splitting and Overlap

In our implementation, the matrix graph splitting is performed without any use of actual topology of underlying physical models. We use the public-domain graph partitioning package METIS [4] to divide the node set into p approximately equal 'subdomains' (or blocks).

The overlap is obtained using the sparsity structure of the qth degree of the coefficient matrix A^q. We refer to q as the overlap size parameter.

4.2 Load Balancing Strategies

Except of relatively small number of additions, an application of the parallel preconditioner BIIC2 (Block Incomplete Inverse Cholesky 2nd order) described in [3] is reduced to p independent lower and upper sparse triangular solves. Here p is the number of parallel processors available.

Even if a nearly equal partition of the matrix into overlapping blocks is used, the quantities NZ(U_t) may be unequal in general. In this case, at the iteration stage the triangular solves become imbalanced which may essentially deteriorate the overall efficiency.

The most obvious load balancing strategy consists in post-filtering of the incomplete factors. Here, one should use a somewhat finer truncation parameter at the factorization stage in order to maintain a sufficient preconditioning quality even with the post-filtered factors.

In what follows, we use the following balancing options:

'STD' denote the preconditioner obtained *as is* with no any post-processing;

'MIN' denotes the post-filtering of each matrix U_t in order to reduce all NZ(U_t) down to the minimum value over p blocks;

'MAX' denotes the augmentation of each U_t with the largest elements taken from the error matrix R_t in order to enlarge all NZ(U_t) up to their maximum value;

'AVR' denotes the combination of the latter two strategies in order to equalize NZ(U_t) near the arithmetic mean value.

5 Numerical Results

We have performed numerical experiments on MVS6000IM computer with Dual Core Intel Itanium 2 processors under RedHat Linux v.2.4.21-20 using MPICH for GM v.1.2.6.14b communication library for data transfer.

5.1 Test Problems and Solution Statistics

We have considered several most ill-conditioned matrices found in the University of Florida sparse matrix collection [8]. The matrix properties are presented in Table 1, where

'Matrix' is the matrix name;

n is the matrix size;

'NZ' is the number of nonzeros in matrix A, and
'Cond' is the estimated condition number.

In all experiments we take the right-hand side $b = Ax^*$, with $x^* \equiv 1$ as the exact solution and $x_0 \equiv 0$ as the initial guess.

In the construction of the preconditioner, the default drop tolerance threshold parameter τ is equal to 10^{-3}. The overlap size parameter q is equal to 10 in all cases.

The PCG iterations were performed until the accuracy $||Ax_k - b||/||b|| < \varepsilon = 10^{-8}$ was achieved.

In the tables below, we use the following notation:

p is the number of processors (blocks) used;
'Balanc' denotes the type of load balancing as indicated earlier in Subsection 4.2;
'Dens' is the preconditioner density $\mathrm{NZ}(U)/\mathrm{NZ}(U_A)$ with respect to the density of the upper triangle of matrix A;
'Imb' is the measure of imbalance in the sizes of preconditioner blocks defined as

$$\mathrm{Imb} = 1 - \frac{\sum_{i=1}^{p} \mathrm{NZ}(U_i)}{p \max_{1 \le i \le p} \mathrm{NZ}(U_i)} \; ;$$

'It' stands for number of iterations, and
'T_{wc}' stands for wall clock total solution time.

5.2 Test Results and Discussion

In the Tables 2–13, we present a comparison for the above mentioned four balancing strategies used with IC(τ), IC(τ^2), and IC2(τ, τ^2) preconditionings on $p = 1$ (serial version) and $p = 8$ processors, obtained for the test problems specified in Table 1.

Table 1. Matrix properties

Matrix	n	NZ	NZ/n	It(PJ)	Cond(A_s)
bcsstk25	15 439	252 241	16.3	3178	3.55×10^{6}
msc23052	23 052	1 154 814	50.0	19086	7.57×10^{7}
gridgena	48 962	512 084	10.4	3216	1.35×10^{5}
cvxbqp1	50 000	349 968	6.9	16	1.73×10^{1}
oilpan	73 752	3 597 188	48.7	18222	1.03×10^{8}
s3dkt3m2	90 449	3 753 461	41.4	40313	3.12×10^{10}
x104	108 384	10 167 624	93.8	64790	2.12×10^{9}
g2_circuit	150 102	726 674	4.8	881	2.34×10^{5}
BenElechi1	245 874	13 150 496	53.4	31501	1.84×10^{9}
msdoor	415 863	20 240 935	48.6	30081	1.95×10^{8}
af_1_k101	503 625	17 550 675	34.8	17321	4.43×10^{7}
parabolic_fem	525 825	3 674 625	6.9	1321	2.10×10^{5}

Table 2. Comparison of balancing strategies for 'bcsstk25' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	2.21	.00	404	3.67
IC(τ)	8	STD	4.07	.44	396	1.31
IC(τ)	8	MIN	1.80	.00	586	1.26
IC(τ^2)	1	STD	14.83	.00	17	2.29
IC(τ^2)	8	STD	29.91	.48	35	1.45
IC(τ^2)	8	MIN	8.14	.00	63	1.10
IC2(τ, τ^2)	1	STD	3.51	.00	50	2.20
IC2(τ, τ^2)	8	STD	5.74	.45	72	1.07
IC2(τ, τ^2)	8	MIN	2.18	.00	866	2.24
IC2(τ, τ^2)	8	AVR	4.71	.00	109	1.11
IC2(τ, τ^2)	8	MAX	9.24	.00	52	1.03

Table 3. Comparison of balancing strategies for 'msc23052' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	1.75	.00	1320	43.32
IC(τ)	8	STD	4.26	.44	1222	15.28
IC(τ)	8	MIN	1.55	.00	3143	17.54
IC(τ^2)	1	STD	12.89	.00	103	28.78
IC(τ^2)	8	STD	19.94	.57	157	15.92
IC(τ^2)	8	MIN	3.96	.00	1651	19.95
IC2(τ, τ^2)	1	STD	2.42	.00	343	25.89
IC2(τ, τ^2)	8	STD	5.59	.45	269	10.37
IC2(τ, τ^2)	8	MIN	1.87	.00	>10000	—
IC2(τ, τ^2)	8	AVR	5.36	.00	429	9.95
IC2(τ, τ^2)	8	MAX	8.44	.00	194	8.87

Table 4. Comparison of balancing strategies for 'gridgena' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	2.76	.00	198	5.89
IC(τ)	8	STD	3.26	.17	220	1.16
IC(τ)	8	MIN	2.57	.00	243	1.17
IC(τ^2)	1	STD	27.62	.00	11	7.80
IC(τ^2)	8	STD	15.05	.22	114	2.02
IC(τ^2)	8	MIN	10.27	.00	115	1.45
IC2(τ, τ^2)	1	STD	6.33	.00	59	7.22
IC2(τ, τ^2)	8	STD	5.88	.21	125	1.24
IC2(τ, τ^2)	8	MIN	4.42	.00	159	1.29
IC2(τ, τ^2)	8	AVR	5.88	.00	134	1.23
IC2(τ, τ^2)	8	MAX	7.34	.00	117	1.30

Table 5. Comparison of balancing strategies for 'cvxbqp1' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	3.95	.00	135	3.81
IC(τ)	8	STD	7.78	.44	189	2.38
IC(τ)	8	MIN	3.08	.00	1358	7.56
IC(τ^2)	1	STD	55.47	.00	19	23.01
IC(τ^2)	8	STD	52.06	.49	67	5.51
IC(τ^2)	8	MIN	10.32	.00	136	3.23
IC2(τ,τ^2)	1	STD	5.85	.00	48	19.53
IC2(τ,τ^2)	8	STD	10.38	.48	75	3.32
IC2(τ,τ^2)	8	MIN	3.44	.00	>10000	—
IC2(τ,τ^2)	8	AVR	9.17	.00	151	3.32
IC2(τ,τ^2)	8	MAX	16.13	.00	70	3.02

Table 6. Comparison of balancing strategies for 'oilpan' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	1.43	.00	1333	116.54
IC(τ)	8	STD	2.38	.31	1454	30.76
IC(τ)	8	MIN	1.27	.00	1566	21.03
IC(τ^2)	1	STD	10.32	.00	90	63.89
IC(τ^2)	8	STD	10.50	.33	98	12.71
IC(τ^2)	8	MIN	4.82	.00	118	9.24
IC2(τ,τ^2)	1	STD	1.82	.00	148	36.51
IC2(τ,τ^2)	8	STD	2.93	.31	110	7.17
IC2(τ,τ^2)	8	MIN	1.54	.00	6350	90.28
IC2(τ,τ^2)	8	AVR	2.93	.00	154	7.54
IC2(τ,τ^2)	8	MAX	3.93	.00	107	7.27

Table 7. Comparison of balancing strategies for 's3dkt3m2' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	1.77	.00	3728	403.38
IC(τ)	8	STD	2.31	.33	3740	82.29
IC(τ)	8	MIN	1.42	.00	3988	56.32
IC(τ^2)	1	STD	19.90	.00	192	199.48
IC(τ^2)	8	STD	12.33	.36	246	27.53
IC(τ^2)	8	MIN	6.82	.00	246	15.05
IC2(τ,τ^2)	1	STD	2.97	.00	247	79.84
IC2(τ,τ^2)	8	STD	3.61	.30	261	12.66
IC2(τ,τ^2)	8	MIN	2.17	.00	758	17.52
IC2(τ,τ^2)	8	AVR	3.61	.00	271	11.22
IC2(τ,τ^2)	8	MAX	5.06	.00	246	12.47

Table 8. Comparison of balancing strategies for ‘x104’ problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	0.88	.00	5180	606.06
IC(τ)	8	STD	1.82	.40	6314	306.90
IC(τ)	8	MIN	0.75	.00	>10000	—
IC(τ^2)	1	STD	6.70	.00	292	247.29
IC(τ^2)	8	STD	13.39	.52	757	265.93
IC(τ^2)	8	MIN	3.55	.00	1084	123.10
IC2(τ, τ^2)	1	STD	1.12	.00	753	167.35
IC2(τ, τ^2)	8	STD	2.28	.40	941	101.21
IC2(τ, τ^2)	8	MIN	0.91	.00	5617	191.48
IC2(τ, τ^2)	8	AVR	2.23	.00	1143	93.49
IC2(τ, τ^2)	8	MAX	3.77	.00	795	95.53

Table 9. Comparison of balancing strategies for ‘g2_circuit’ problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	4.33	.00	86	6.33
IC(τ)	8	STD	5.24	.10	124	1.82
IC(τ)	8	MIN	4.05	.00	134	1.77
IC(τ^2)	1	STD	54.87	.00	11	32.82
IC(τ^2)	8	STD	48.28	.12	26	4.42
IC(τ^2)	8	MIN	25.84	.00	25	3.80
IC2(τ, τ^2)	1	STD	6.69	.00	26	18.85
IC2(τ, τ^2)	8	STD	7.74	.10	38	2.84
IC2(τ, τ^2)	8	MIN	5.88	.00	54	2.82
IC2(τ, τ^2)	8	AVR	7.74	.00	37	2.68
IC2(τ, τ^2)	8	MAX	8.56	.00	36	2.69

Table 10. Comparison of balancing strategies for ‘BenElechi1’ problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	1.79	.00	2450	985.59
IC(τ)	8	STD	2.18	.13	2535	169.36
IC(τ)	8	MIN	1.70	.00	2544	137.06
IC(τ^2)	1	STD	17.19	.00	152	558.59
IC(τ^2)	8	STD	14.03	.26	295	117.34
IC(τ^2)	8	MIN	9.30	.00	295	88.52
IC2(τ, τ^2)	1	STD	2.59	.00	276	295.10
IC2(τ, τ^2)	8	STD	3.04	.12	382	57.41
IC2(τ, τ^2)	8	MIN	2.32	.00	597	65.23
IC2(τ, τ^2)	8	AVR	3.04	.00	397	56.47
IC2(τ, τ^2)	8	MAX	3.47	.00	361	56.11

Table 11. Comparison of balancing strategies for 'msdoor' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	2.00	.00	1905	1275.95
IC(τ)	8	STD	2.50	.22	1972	247.80
IC(τ)	8	MIN	1.93	.00	1999	186.61
IC(τ^2)	1	STD	out	of	memory	—
IC(τ^2)	8	STD	17.40	.20	178	156.57
IC(τ^2)	8	MIN	10.77	.00	181	121.82
IC2(τ, τ^2)	1	STD	2.62	.00	198	495.71
IC2(τ, τ^2)	8	STD	3.23	.23	202	76.51
IC2(τ, τ^2)	8	MIN	2.42	.00	361	84.40
IC2(τ, τ^2)	8	AVR	3.24	.00	234	76.04
IC2(τ, τ^2)	8	MAX	4.23	.00	191	75.67

Table 12. Comparison of balancing strategies for 'af_1_k101' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	1.32	.00	908	405.86
IC(τ)	8	STD	1.45	.12	1023	72.73
IC(τ)	8	MIN	1.28	.00	1025	67.58
IC(τ^2)	1	STD	12.61	.00	60	254.40
IC(τ^2)	8	STD	9.74	.13	139	47.25
IC(τ^2)	8	MIN	6.91	.00	140	38.29
IC2(τ, τ^2)	1	STD	1.95	.00	163	170.64
IC2(τ, τ^2)	8	STD	2.01	.11	212	26.87
IC2(τ, τ^2)	8	MIN	1.77	.00	384	38.30
IC2(τ, τ^2)	8	AVR	2.01	.00	239	28.59
IC2(τ, τ^2)	8	MAX	2.25	.00	197	26.34

Table 13. Comparison of balancing strategies for 'parabolic_fem' problem

Method	p	Balanc	Dens	Imb	It	T_{wc}
IC(τ)	1	STD	3.24	.00	138	41.18
IC(τ)	8	STD	3.53	.15	214	9.86
IC(τ)	8	MIN	2.70	.00	231	8.83
IC(τ^2)	1	STD	42.60	.00	11	89.48
IC(τ^2)	8	STD	28.42	.13	70	17.84
IC(τ^2)	8	MIN	23.97	.00	70	16.02
IC2(τ, τ^2)	1	STD	5.29	.00	62	40.79
IC2(τ, τ^2)	8	STD	5.86	.12	97	7.72
IC2(τ, τ^2)	8	MIN	5.01	.00	108	7.67
IC2(τ, τ^2)	8	AVR	5.86	.00	100	7.63
IC2(τ, τ^2)	8	MAX	6.63	.00	93	7.57

It is seen that the most obvious load balancing based on the 'MIN' strategy often shows poor convergence. For instance, for the problem 'msc23052' the convergence is not achieved for 10000 iterations (see Table 3).

The sharp increase in the iteration number for many hard-to-solve problems (such as 'bcsstk25', 'msc23052', 'oilpan', 'cvxbqp1', 'x104') for BIIC2 preconditioning with 8 processors and 'MIN' balancing strategy is explained by the presence of very large $\lambda_{\max}(\tilde{M})$, as indicate the estimated bounds for this quantity. Note that typical values which guarantee the numerical stability of the PCG iterations are below 2. This destructive effect may be related to the violation of the upper bound on the norm of truncated parts of U_t, see Theorem 2 earlier.

On the other hand, the numerical results show that the proposed load balancing based on the 'MAX' strategy is sufficiently robust and efficient. In many cases, the use of 'MAX' balancing strategy resulted in a smaller total solution time as compared to the original (unbalanced) 'STD' version.

References

1. Kaporin, I.E.: New convergence results and preconditioning strategies for the conjugate gradient method. Numer. Linear Algebra Appls. 1, 179–210 (1994)
2. Kaporin, I.E.: High quality preconditionings of a general symmetric positive definite matrix based on its $U^TU + U^TR + R^TU$-decomposition. Numer. Lin. Alg. Appl. 5, 483–509 (1998)
3. Kaporin, I.E., Konshin, I.N.: A parallel block overlap preconditioning with inexact submatrix inversion for linear elasticity problems. Numer. Lin. Algebra Appl. 9, 141–162 (2002)
4. Karypis, G., Kumar, V.: Multilevel k-way hypergraph partitioning, Technical Report 98-036, Dept. Comp. Sci. Engrg. Army HPC Research Center, Univ. of Minnesota, MN (1998)
5. Jennings, A., Malik, G.M.: Partial elimination. J. Inst. Math. Appl. 20, 307–316 (1977)
6. Manteuffel, T.A.: An incomplete factorization technique for positive definite linear systems. Math. Comput. 34, 473–497 (1980)
7. Notay, Y.: On the convergence rate of the conjugate gradients in presence of rounding errors. Numer. Math. 65, 301–317 (1993)
8. University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices`

Distributions and Schedules of CPU Time in a Multiprocessor System When the Users' Utility Functions Are Linear

Alexander Khutoretskij[1] and Sergei Bredikhin[2]

[1] Novosibirsk State Teacher's Training University
hab@dus.nsc.ru
[2] Institute of Computational Mathematics and Mathematical Geophysics SB RAS
bred@nsc.ru

Abstract. We consider a market of CPU time in a multiprocessor system providing the payable services. We suggest a market model under the following assumptions: (i) processors may be of different capacity; (ii) the processors' owners (suppliers) may have the different reservation prices; (iii) every user (consumer) has a linear utility function and one task within the considered period; (iv) a task can use different processors, but it cannot use two processors at the same moment. The main result is that all equilibrium CPU time distributions can be obtained as solutions to some linear programming problem, and equilibrium prices can be determined on the base of shadow prices for the same problem. An equilibrium distribution shows, what time each task should use each processor, but, generally speaking, it doesn't determine a scheduling. If the most labour-intensive job can be completed within the time-period to be distributed, using only the slowest processor, then the simple algorithm exists for constructing a scheduling corresponding to some equilibrium distribution. In general case such algorithm is unknown.

Keywords: Distribution, scheduling, multiprocessor system, CPU time, paid services, market equilibrium, linear programming.

1 Introduction

With an extension of the circle of users and development of computing systems, there arises the problem of funding the operating costs and capital spendings [1, 2]. The concept of computing systems rendering paid services is very popular now. The problem of financing is not resolved still for such systems. It would be ideal to incorporate the subsystem calculating the equilibrium prices into the resource management mechanism. Then the resources could be sold at calculated equilibrium prices, and the problem of paid services were solved. Models of equilibrium in the market of CPU time were considered in [3 – 5]. In these papers, the prices of equilibrium are derived without taking into account possible differences in processors' throughputs.

2 Market Model

Market structure. We consider the market of cycles of different processors within the period of length T. There are M customers (users) and N suppliers in the market. Each supplier owns one processor, these processors may have various throughputs, and suppliers may have various reservation prices. Each user has a linear utility function, and during considered period he would like to execute one task. The task may use various processors, but not two processors simultaneously.

Model of a supplier. The supplier j owns a processor which performs s_j cycles during a time unit with specific cost c_j. During the period it may provide total s_jT cycles.

When selling q_j CPU cycles at the price p_j, the supplier will obtain the profit $(p_j - c_j)q_j$. His goal is to choose supply during the period so as to maximize the profit (supplier's surplus): $(p_j - c_j)q_j \to$ max subject to $0 \le q_j \le s_jT$.

Then the supply correspondence of supplier j has the following form:

$$S_j(p_j) = \begin{cases} 0, \text{ if } p_j < c_j, \\ [0, s_jT\,], \text{ if } p_j = c_j\,, \\ s_jT, \text{ if } p_j > c_j. \end{cases} \tag{1}$$

Model of a user. The user i has budget B_i and one task of volume Q_i (CPU cycles). Therefore, he is willing to pay $a_i = B_i \,/\, Q_i$ for one cycle.

Consumption bundle of the user i is a vector $x^i = (x_{ij})_j$, where $x_{ij} \ge 0$ is the number of cycles of processor j allocated to task i. Such vector x^i should meet the following conditions: (a) $\sum_j (x_{ij} / s_j) \le T$ (the total execution time of a task over all processors should not exceed the length of period); (b) $\sum_j x_{ij} \le Q_i$ (the total number of cycles allocated to a task at all processors should not exceed its volume).

Let X_i be the set of all consumption bundles of the user i and let p_j be the price of the processor j cycles. Set $p = (p_j)_j$. Choosing some $x^i \in X_i$ under the prices p**,** the user i obtains utility (consumer's surplus) $v_i(x^i) = \sum_j (a_i - p_j)x_{ij}$. Thus, the user i selects $x^i \in X_i$ so as to maximize utility function $v_i(x^i)$ under the budget restriction $\sum_j p_j x_{ij} \le B_i$.

3 Equilibria

Each supplier j owns s_jT CPU cycles during the considered period. If he have chosen supply q_j, then the situation may be equivalently interpreted as follows: the supplier j demanded $x_{0j} = s_jT - q_j$ cycles of the processor j. Set $x^0 = (x_{0j})_j$.

Equilibrium in the considered market is a pare (x, p), where $x = (x^0, x^1, \ldots, x^n)$, such that: (i) $\sum_{i=0}^{N} x_{ij} \le s_jT$ for all j with equality if $p_j > 0$ (the excess supply has zero price); (ii) x^i maximizes the utility function $v_i(x^i)$ over $x^i \in X_i$; (iii) $p_i \ge 0$ for all i.

If (x, p) is an equilibrium, then p is a *vector of equilibrium prices*, and x is an *equilibrium distribution*.

Proposition 1. If (y, p) is an equilibrium, then there exists such equilibrium (x, p) with the same prices, that

$$\sum_{i=0}^{N} x_{ij} = s_j T \quad \text{for all } j, \tag{2}$$

i.e. at the same equilibrium prices an equilibrium distribution exists, in which the demand of each supplier is exactly equal to unused remainder of the corresponding resource.

Proposition 2. If (x, p^1) is an equilibrium, then there exists such equilibrium (x, p) with the same distribution, that

$$p_j \geq c_j \quad \text{for all } j. \tag{3}$$

We shall call equilibrium (x, p) *normal*, if it satisfies the conditions (2) and (3).

On the basis of the propositions 1 and 2 we will consider only normal equilibria later on. In order to describe all such equilibria, let us consider the following linear programming problem:

$$f(x) = \sum_{i=1}^{M} \sum_{j=1}^{N} (a_i - c_j) x_{ij} \to \max \tag{4}$$

subject to

$$\sum_{j} x_{ij} \leq Q_i \quad \text{for all } i \geq 1, \tag{5}$$

$$\sum_{i=0}^{N} x_{ij} = s_j T \quad \text{for all } j, \tag{6}$$

$$\sum_{j} \frac{x_{ij}}{s_j} \leq T \quad \text{for all } i \geq 1, \tag{7}$$

$$x \geq 0. \tag{8}$$

The problem dual to (4) – (8) has the following form:

$$h(\alpha, \pi, \beta) = \sum_{i \geq 1} \alpha_i Q_i + T \sum_{j} \pi_j s_j + T \sum_{i \geq 1} \beta_i \to \min \tag{9}$$

$$\text{subject to:} \quad \alpha_i + \pi_j + \beta_i / s_j \geq a_i - c_j; \quad \alpha_i \geq 0; \quad \pi_j \geq 0; \quad \beta_i \geq 0 \ \text{ for all } i, j. \tag{10}$$

Here $\alpha = (\alpha_i)_i$, $\pi = (\pi_j)_j$ and $\beta = (\beta_i)_i$ are vectors of dual variables for restrictions (5), (6), and (7), respectively.

The following two theorems establish a correspondence between the normal equilibria and optimal solutions to problems (4) – (8), (9) – (10).

Theorem 1. If x is an optimal solution to the problem (4) – (8), (α, π, β) is an optimal solution to the problem (9) – (10), and $p_j = c_j + \pi_j$, then (x, p) is a normal equilibrium.

Theorem 2. If (x, p) is a normal equilibrium and $\pi_j = p_j - c_j$ for all j, then x is an optimal solution to the problem (4) – (8) and there exist such vectors α and β, that (α, π, β) is an optimal solution to the problem (9) – (10).

Let us note some properties of a normal equilibrium (x, p).

1. $\sum_j p_j x_{ij} \le B_i$ (in equilibrium, the total payment of each user does not exceed his budget).

2. If $x_{ij} > 0$, then $c_j \le p_j \le a_i$ (the equilibrium price determines the distribution of the total surplus $c_j - a_i$ between consumer i and supplier j).

3. If $s_j < s_k$ and $x_{ij} > 0$ for some i, then $p_j < p_k$ (the less throughput of an active processor, the less price of its cycle).

4 Schedules

An optimal solution to the problem (4) – (8) provides an equilibrium distribution of CPU time. A question appears: is it possible to realize this distribution by any schedule in the sense of the following definition.

The schedule corresponding to distribution x is a collection of intervals $[\alpha_{ij}^k, \beta_{ij}^k]$ (during which execution of the task i is scheduled at the processor j) for each i, j $(1 \le i \le M,\ 1 \le j \le N)$ such, that: (a) $[\alpha_{ij}^k, \beta_{ij}^k] \cap [\alpha_{mn}^l, \beta_{mn}^l] = \varnothing$ if: $j = n$ and $i \ne m$; or $j = n$, $i = m$, and $k \ne l$; or $j \ne n$ and $i = m$ (intervals allocated to different tasks at the same processor are pairwise disjoint, and intervals allocated to the same task are pairwise disjoint as well); (b) $s_j \sum_k [\beta_{ij}^k - \alpha_{ij}^k] = x_i^j$ (the total time allocated to task i at processor j covers x_i^j cycles). The schedule corresponding to an equilibrium distribution (if any) will be called *equilibrium schedule*.

4.1 A Sufficient Condition for Existence of an Equilibrium Schedule

Assume that the most laborious task may be solved at the slowest processor during the considered period:

$$Q_i \le s_j T \quad \text{for all } i, j. \tag{11}$$

Assumption (11) simplifies the problem (4) – (8) making the constraints (7) unnecessary: they are fulfilled for any distribution that meets other restrictions of the problem.

Assume that the processors are numbered in the order of non-decreasing c_j, and the tasks are numbered in the order of non-increasing a_i.

Let us say that the distribution x is *dense*, if

$$x_{mn} = \min\{s_n T - \textstyle\sum_{i<m} x_{in},\ Q_m - \textstyle\sum_{j<n} x_{jm}\} \quad \text{for all } m, n. \tag{12}$$

It is easy to see that the dense distribution is uniquely defined by the orderings of the values a_i and c_j.

Theorem 3. Under the condition (11), the dense distribution is an optimal solution of the problem (4) – (8) (and, consequently, it is an equilibrium distribution).

Theorem 4. Under the condition (11), there exists a schedule corresponding to the dense distribution (and, consequently, it is an equilibrium schedule).

The last theorem is proven constructively, i.e. the simple algorithm is proposed for construction of the schedule corresponding to the dense distribution.

4.2 Sketch of the Algorithm for Constructing the Equilibrium Schedule

If $a_1 < c_1$, then the equilibrium schedule is empty, none of tasks is solved.

If $a_1 \geq c_1$, then at each step we will do the following: select the first underloaded processor j and the first incompletely allocated task i. If $a_i \geq c_j$, then allocate to task i at processor j a time-interval with beginning at the first moment unoccupied at this processor. The length of this interval is defined by (12): the minimum of non-allocated volume of the task and non-distributed capacity of the processor. If $a_i < c_j$, or all tasks are allocated, or all processors are loaded, then the schedule is constructed.

From assumption (11) it follows that: (a) the algorithm will allocate not more than two intervals (possibly, no intervals) to each task; (b) if two intervals are allocated to the same task, then they do not intersect, and are located at processors with consecutive numbers (j and $j + 1$); one of this intervals occupies the end of the distributed period at the processor j, and the second one occupies the beginning of the period at the processor $j + 1$.

References

1. Gray, J.: Distributed Computing Economics. In: Herbert, A.J., Jones, K.S. (eds.) Computer Systems: Theory, Technology, and Applications, pp. 93–101. Springer, New York (2003)
2. Wolski, R., Brevik, J., Plank, J., Bryan, T.: Grid resource allocation and control using computational economies. In: Berman, F., Fox, G., Hey, T. (eds.) Grid Computing. Making the Global Infrastructure a Reality, pp. 747–773. John Wiley & Sons, Chichester (2003)
3. Bredin, J., Kotz, D., Rus, D.: Utility Driven Mobile-Agent Scheduling. Technical Report PCS-TR98-331. Hanover, Dartmouth College, Department of Computer Science (1998)
4. Bredikhin, S.V., Vyalkov, I.A., Savchenko, I., Yu., K.A.B.: Two Models of Adjusting for Distribution of Computational Resources (in Russian). Siberian Journal of Industrial Mathematics, IX(1(25)), 28–46 (2006)
5. Bredikhin, S.V., Tiunova, E.A., Khutoretskij, A.B.: Price Coordination of Supply and Demand at Distribution of Multiprocessor System Capacity (in Russian). Siberian Journal of Industrial Mathematics X(1(31)), 20–28 (2007)

Visualizing Potential Deadlocks in Multithreaded Programs*

Byung-Chul Kim[1], Sang-Woo Jun[2], Dae Joon Hwang[3], and Yong-Kee Jun[1,**]

[1] Gyeongsang National University, Jinju, 660-701, South Korea
[2] Seoul National University, Seoul, 151-742, South Korea
[3] SungKyunKwan University, Suwon, 440-746, South Korea
bckim@gnu.ac.kr, aradia.jun@gmail.com, djhwang@skku.edu, jun@gnu.ac.kr

Abstract. It is important to analyze and identify potential deadlocks resident in multithreaded programs from a successful deadlock-free execution, because the nondeterministic nature of such programs may hide the errors during testing. Visualizing the runtime behaviors of locking operations makes it possible to debug such errors effectively, because it provides intuitive understanding of different feasible executions caused by nondeterminism. However, with previous visualization techniques, it is hard to capture alternate orders imposed by locks due to their representation of a partial-order over locking operations. This paper presents a novel graph, called *lock-causality graph*, which represents alternate orders over locking operations. A visualization tool implements the graph, and demonstrates its power using the classical dining-philosophers problem written in Java. The experiment result shows that the graph provides a simple but powerful representation of potential deadlocks in an execution instance not deadlocked.

Keywords: multithreaded programs, debugging, potential deadlocks, visualization, lock-causality graph.

1 Introduction

A lock is a synchronization mechanism for preventing shared resources from being accessed by threads with no proper synchronization. A deadlock [1,4], which blocks some threads permanently, is one of the most common problems in multithreaded programs [5,15] which use the lock mechanism. But the inherent nondeterminism of multithreaded programs makes these deadlocks difficult to analyze, test and debug [9,13]. Nondeterminism makes multiple executions of same programs with the same input produce different results. This means that

* This research was supported by the MKE (The Ministry of Knowledge Economy), Korea,under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute for Information Technology Advancement) (IITA-2009-C1090-0904-0001).

** Corresponding author: In Gyeongsang National University, he is also involved in the Research Institute of Computer and Information Communication (RICIC).

deadlocks resident in such programs may never occur during testing. Therefore, debugging of multithreaded programs requires an effective technique to analyze and identify potential deadlocks from a successful deadlock-free execution.

Some researches [1,2,4] have presented automatic techniques for detecting potential deadlocks from an observed execution which is not deadlocked. These techniques report the location of detected potential deadlock with the information about threads and locks involved in such deadlocks, but do not provide the information on what executions can actually cause such deadlocks. Therefore, the results reported by the automatic techniques allow users to check whether a program may contain deadlocks, but they are still difficult to debug the detected deadlocks correctly and reasonably.

In general,the execution of multithreaded programs is complex as well as the amount of traces is large, so that it is not easy to analyze the traces for debugging such errors. Visualization [7,10,13] can ease the understanding of the large and complex situations. Visualizing the runtime behaviors of locking operations helps to debug such errors correctly and reasonably, because it provides intuitive understanding of different feasible executions caused by nondeterminism. However, The previous visualization techniques [3,6,12,14,16] represent a partial-order over locking operations with no reflection of alternate orders which could have occurred by such operations.

This paper presents a novel graph, called *lock-causality graph*, which represents alternate orders over locking operations for helping to identify and debug potential deadlocks in multithreaded programs. The graph uses a set of traces to represent locking operations as its nodes, and threading operations between two code blocks as its edges. The constructed graph has three types of symbols, which are $\sqcap$, $\sqcup$, and $\vee$ to depict the locking operations and three kinds of arrow symbols, which are solid, dashed, and dotted arrows to depict the threading operations. A visualization tool implements the graph, and demonstrates its power using the classical dining-philosophers problem written in Java. The experiment result shows that the graph provides a simple but powerful representation of potential deadlocks in a successful deadlock-free execution.

This paper has some limitations. First, this paper only attempts to depict the executions caused by different schedules, not different inputs to the program. This limits is intrinsic to pure dynamic analysis, which look only at executions and not at the program itself. Second, this paper helps to identify potential deadlocks in multithreaded programs, but does not detect such errors automatically. The users' effort is required for identifying potential deadlocks from the visualized behavior and is essential in analyzing the cause of such errors for debugging.

This paper is organized as follows: Section 2 introduces previous work which visualizes the behavior of locking operations for debugging multithreaded programs. In Section 3, visualizing potential deadlocks with the lock-causality graph is presented. Section 4 describes a visualization tool which implements the graph. Finally, conclusion and future works on this paper are given in Section 5.

2 Related Work

Many approaches have developed for visualizing the runtime behavior of locking operations for debugging deadlocks in multithreaded programs. Some of these approaches, Javis [14] and Jacot [12], use the UML (Unified Modeling Language) paradigm which is the standard for visual modeling of object-oriented systems. Javis is a visualizing and debugging tool for multithreaded Java programs. Javis visualizes only the actual deadlocks detected from traces on two diagrams, which are the sequence diagram and the collaboration diagram. Javis supports abstraction, showing only the objects directly involved in the deadlock. Jacot is also a dynamic visualization tool for multithreaded Java programs. The tool has the UML sequence diagram and the thread state diagram to depict the interaction between objects and the interleaving of threads over the flow of time.

However, the standard sequential diagram of UML is insufficient for representing the happens-before relation of multithreaded programs [3]. An extension technique [3] of the sequential diagram is presented in order to address shortcomings of such a diagram. The technique adds some notations for threads as executable tasks and the happens-before relation between events.

MutexView [16] has two size-varying circles to represent the relationship between threads and locks of the POSIX thread programming library on the KSR system. A thread is indicated with a small circle distinguished by color and the locks are represented with big circles. When a thread is trying to acquire the lock, the small circle of the thread appears somewhere around the circle corresponding to the lock. When a thread gets the lock, it moves into the circle and remains there until it releases the lock. Then it leaves the circle and disappears. MutexView animates such situations and represents the actual deadlocks at runtime.

The History Graph Window [6] shows the execution history of all threads. This tool shows each thread with a history bar running left to right. Each history bar has a color coded with green, blue, or red for running, joining or blocked by a synchronization primitive, respectively. The History Graph Window has several tags, each of which is related to a monitor, to visualize the behavior of locking operations on-the-fly.

These approaches are hard to analyze and identify potential deadlocks from their representations of multithreaded programs. They reflect a partial-order rather than alternate orders over locking operations. This makes it difficult to analyze different feasible executions which could have occurred by locking operations.

3 Visualizing Potential Deadlocks

In order to visualize potential deadlocks in multithreaded programs, we presents the lock-causality graph which represents alternate orders over locking operations. The lock-causality graph uses a set of traces from a successful deadlock-free execution, which collects threading operations and locking operations. The

```
// Trace for        // Trace for      //  Trace for       // Trace for
// Main thread      // t1 thread      // t2 thread        // t3 thread
1:TS:0:0:0:0        1:TS:0:0:1:2      1:TS:0:0:1:3        1:TS:0:0:1:2
2:TC:0:0:0:0        2:TC:0:0:0:0      2:LA:1:L1:0:0       2:LA:1:L3:0:0
3:TC:0:0:0:0        3:LA:1:L1:0:0     3:LA:2:L3:0:0       3:LN:1:L3:0:0
4:TJ:0:0:1:9        4:LA:2:L2:0:0     4:LA:3:L2:0:0       4:LA:2:L2:0:0
5:LA:1:L3:0:0       5:LA:3:L3:0:0     5:LW:3:L3:0:0       5:LR:1:L2:0:0
6:LA:2:L2:0:0       6:LR:2:L3:0:0     6:LR:2:L2:0:0       6:LR:0:L3:0:0
7:LR:1:L2:0:0       7:LR:1:L2:0:0     7:LR:1:L3:0:0       7:TT:0:0:0:0
8:LR:0:L3:0:0       8:LR:0:L1:0:0     8:LR:0:L1:0:0
9:TT:0:0:0:0        9:TT:0:0:0:0      9:TT:0:0:0:0
```

Fig. 1. A set of deadlock-free traces generated from an execution of a multithreaded program

threading operations refer to thread start (TS), thread terminate (TT), thread create (TC), and thread join (TJ) events. The TS and the TT events occur when a thread starts and finishes the execution of its *run* method, respectively. The TC event is generated when a thread completes the invocation of the *start* method on a new thread, and the TJ event is generated when a thread, blocked by invoking *join* method on another thread, resumes its execution. The locking operations refer to lock acquire (LA), lock release (LR), lock wait (LW), lock notify (LN) events. The LA and LR events are generated when a thread obtains and releases a lock for entering and leaving a synchronized region, respectively. The LW event occurs when a thread executing a synchronized region invokes the *wait* method on an object. The LN event occurs when a thread invokes the *notify* method on an object.

Fig. 1 shows a set of deadlock-free traces collected from an execution of a multithreaded program. Each thread may have a local-file to store its trace for efficiency. Each line of the trace consists of six entries separated by ':'. The first entry is a timestamp indicating the number of events a thread has generated by that time. This number is increased by one whenever a thread generates an event. The second entry is the type of the events generated in the thread. The third entry is the locking level indicating the number of locks owned by the thread at that time. The fourth entry indicates the identifier of an object specified by the locking operations. The next two entries consist of a thread identifier and a timestamp, indicating the event which corresponds to this event.

The collected traces are used to analyze code blocks and execution orders over the code blocks. A code block is defined to be a set of instructions between two consecutive events of interest in the same thread. Each of Main, t1, and t2 threads has 8 code blocks and the thread t3 has 6 code blocks. The execution order is decided according to event types. If an event type is one of TS, TT, TC, and TJ events, which are threading operations, the execution order defined by the event is "deterministic". Deterministic order means that the blocks defined

```
// Nodes for          // Nodes for          // Nodes for          // Nodes for
// Main thread        // t1 thread          // t2 thread          // t3 thread
0,1:-,-:-,-:0         1,1:-,-:-,-:0         2,1:-,-:-,-:0         3,1:-,-:-,-:0
0,2:-,-:-,-:0         1,2:-,-:-,-:0         2,2:LA,-:L1,-:1       3,2:LA,LN:L3,L3:1
0,3:-,-:-,-:0         1,3:LA,-:L1,-:1       2,3:LA,-:L3,-:2       3,3:-,-:-,-:1
0,4:-,-:-,-:0         1,4:LA,-:L2,-:2       2,4:LA,-:L2,-:3       3,4:LA,LR:L2,L2:2
0,5:LA,-:L3,-:1       1,5:LA,LR:L3,L3:3     2,5:LW,LR:L3,L2:3     3,5:-,LR:-,L3:1
0,6:LA,LR:L2,L2:2     1,6:-,LR:-,L2:2       2,6:-,LR:-,L3:2       3,6:-,-:-,-:0
0,7:-,LR:-L3:1        1,7:-,LR:-,L1:1       2,7:-,LR:-,L1:1
0,8:-,-:-,-:0         1,8:-,-:-,-:0         2,8:-,-:-,-:0
```

(a) The code blocks including the nondeterministic orders

```
(TC:0,1:1,1), (TC:0,2:2,1), (TC:1,1:3,1), (TJ:1,8:1,4)
```

(b)The deterministic orders

Fig. 2. Information generated by analyzing the traces of Fig. 1. The code blocks and the deterministic orders are represented as nodes and edges of a lock-causality graph, respectively.

by the event always execute in the same order in different executions. Otherwise, i.e. if the event type is one of the locking operations, the execution order caused by the event is "nondeterministic." Nondeterministic order means that the blocks defined by the event may occur in different order in different executions.

Fig. 2 shows the code blocks and the execution orders generated after the analysis of the traces. The nondeterministic orders are shown on the code blocks and the deterministic orders are shown separately. The code blocks consist of four entries separated by ':'. The first entry indicates a unique identifier of a block, which consists of a thread identifier and a block number. The second entry, which consists of two locking operations, indicates the nondeterministic order constrained on the block. The mark '-' implies that the block does not have a locking constraint. The next entry specifies two identifiers of objects used by the locking operations. (The identifier of the object is a positive number rather than the symbolic name in practice.) The mark '-' implies that this block does not have such an object. The last entry specifies the number of locks which the block owns. A deterministic order consists of a operation type, a source block identifier, and a destination block identifier. The entries of each deterministic order are separated by ':'.

The lock-causality graph is a directed-acyclic graph whose nodes represent the code blocks and whose edges represent the deterministic orders. The construction of the graph is performed in two steps. In the first step, the nodes in the same thread are connected and in the second step, two nodes in different threads are connected by the deterministic order. In order to visualize the constructed graph, we simply symbolize a node with a rectangle box and an edge with an arrow. The constraints on a block imposed by nondeterministic orders are marked by three symbols, which are $\sqcap$ for LA event, $\sqcup$ for LR event, and $\vee$ for LW and LN events. Their corresponding locking objects are distinguished by a unique color. The graph has three kinds of edges, which are sequential edges for connecting two nodes in the same thread, fork edges for TC events, and join edges for TJ events. These three edges are represented with a solid arrow for a sequential edge, a dotted arrow for a join edge, and a dashed arrow for a fork edge with the head forward the destination node of an edge. Fig. 3 shows the visualized lock-causality graph which was produced by the code blocks and the deterministic orders of Fig. 2. The graph uses three colors for representing three lock objects, green for the L1 object, red for the L2 object, and blue for the L3 object.

4 Debugging Potential Deadlocks

Nondeterminism may result in deadlocks resident in multithreaded programs that are hidden during testing. The lock-causality graph presented by this paper aims at helping to analyze and identify potential deadlocks that have not occurred in an observed execution. This section describes how users analyze and identify potential deadlocks on the graph.

The lock-causality graph captures the happens-before relation over locking operations and locking constraints imposed on the blocks. The happens-before relation defines a partial order between the locking operations. If a locking operation l_1 happens before another locking operation l_2, then l_1 must occur before l_2 in all feasible executions of the program with the same input. Determining such a relation on two locking operations requires checking whether there is a path between them on the graph. A locking operation l_1 is an ancestor of another locking operation l_2 if there is a path from l_1 to l_2 on the graph. Two locking operations are concurrent if neither one is an ancestor of the other. The locking constraint imposed on a block means that the block can not be executed by multiple threads at the same time. A sequence of blocks, which starts from the symbol $\sqcap$ and ends at its closest symbol $\sqcup$ with the same color, defines a synchronized region. When a thread is being executed in the region, other threads attempting to enter another synchronized region defined by the same color are blocked.

The graph of Fig. 3 shows that thread t1 has tried to acquire two locks in order of red and blue colors, while the others have obtained them in order of blue and red colors. The situation implies that three pairs of threads, (t1, Main), (t1, t2), and (t1, t3), have a possibility to be involved in deadlocks. However, the additional information perceived from the graph tells that t1 and Main threads are not involved in a deadlock, because the join edge from t1 to Main makes the synchronized region in Main thread to be always executed after t1 thread's execution is completed. The graph shows that t1 and t2 threads do not produce such an error. That is because the synchronized regions in both threads are protected by another locking object with green color. In order for both threads to enter the synchronized regions, they must obtain the lock for the object protecting the regions first. The graph intuitively shows that there is no path between the

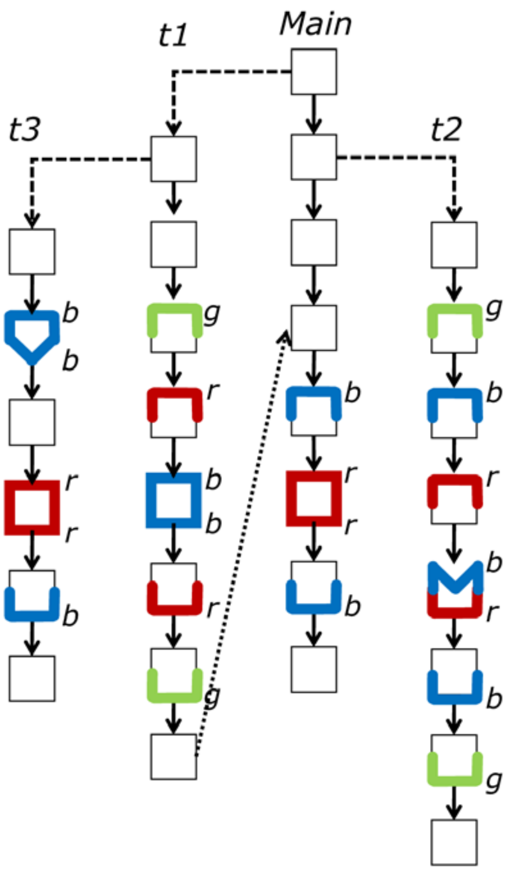


Fig. 3. The lock-causality graph visualized with the nodes and edges of Fig. 2. The small letters, r, g, and b right to the code blocks indicate red, blue, and green, respectively.

synchronized regions in `t1` and `t3` nor any locking object protecting them on the graph. Therefore, two threads may require their second locks while owning their first locks in some executions. Such a situation results in both threads involved in a deadlock.

The lock-causality graph provides some intuitive ways to debug the discovered potential deadlock. The first way is to wrap the synchronized region in `t1` thread with the green-colored object. This is the simplest way but the extension of synchronized region may decrease the performance of the program. The second way is to change the order of the red and blue locks acquired by the `t1` thread into the order of the locks of blue and red. Users can easily understand that the second way is the clearer and more reasonable way to debug the potential deadlock without performance loss, compared to the first treatment.

5 Tool Development

We have implemented a tool that uses the lock-causality graph to visualize the behavior of locking operations for analyzing and identifying potential deadlocks in multithreaded Java programs. The tool consists of two modules: collector and visualizer. The collector is the module in charge of capturing the runtime information during the execution of a program and then storing the information on thread-local files for efficiency. The input of the collector is a Java program with a main method and a list of classes the user is interested in. The collector captures the runtime information of objects, classes and threads related to the given classes. That is, the module filters out information which have no relationship to the given classes.

The visualizer module analyzes the collected information, constructs the lock-causality graph, and then draws the graph on a planar view. This module needs to decide the position of the nodes in the graph. Main thread is located at the horizontal center of the view and the location of the threads created by the Main thread follows the odd-even system. For example, the first thread created by the Main thread is positioned on the left side of the Main thread and the second thread is placed on its right side. The latest created thread has the nearest position to its parent. The nodes in a thread are assigned to a single vertical line. The vertical position of a node is positioned below more than that of its parent node. The vertical position of the first node in the Main thread is always zero. The vertical position of the first nodes in the other threads is always one more in value than that of the source node of the fork edge.

The tool is applied to the classical "dining-philosophers problem" for experimenting the effectiveness of the lock-causality graph. The experiment is performed on a Windows system with Intel Dual 2.5 GHz processor and 2 GB memory. We suppose that all five philosophers take their left chopsticks simultaneously. Then they will all block on their right chopsticks, and there will be a deadlock.

Fig. 4 shows the result of our tool applied on the program. The tool has four panels, which are a static information panel, a dynamic information panel, a graph panel, and a message panel to provide information related to the programs of interest. The static information panel has a directory view for users to register classes they are interested in and execute a Java program, and a class view for showing the registered classes. The dynamic information panel has an object view and a thread view for showing information of objects and threads, which are instances of the classes of interest, respectively. The graph panel shows a lock-causality graph for representing the runtime behavior of the program of interest. Finally, the message panel shows output and error messages generated during execution in the output view and the error view, respectively.

In Fig. 4, the class view implies that the program was executed with three registered classes of interest and the object view tells us that five objects were instantiated from the class Chopstick and five objects were instantiated from the class Philosopher. The graph view shows that a thread creates five other threads being synchronized to other threads with two locking objects, and then

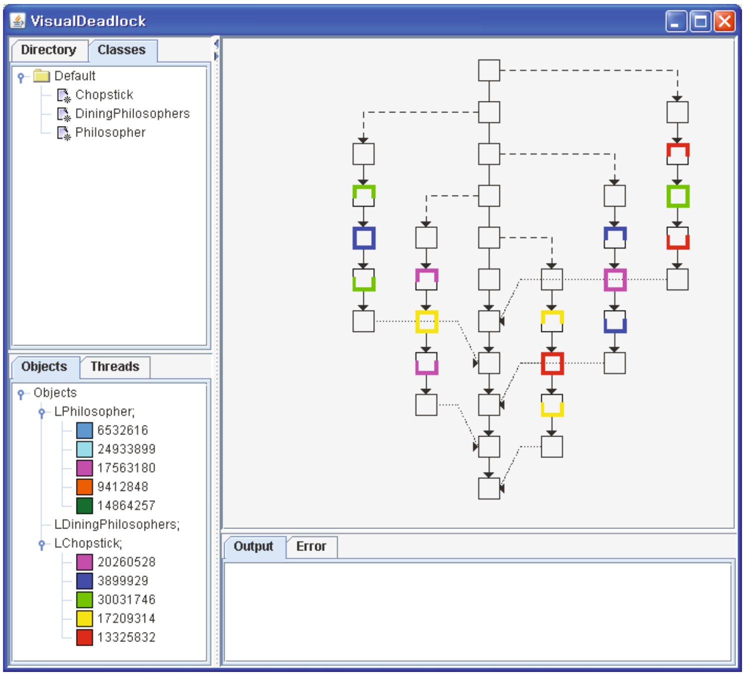


Fig. 4. The tool shows from an successful deadlock-free execution that the experimented program can be deadlocked in different scheduling cases

waits for each created thread to complete. The graph view indicates that all of the locking objects are assigned to each thread, except for one thread, and then simultaneously assigned to the other threads. Therefore, users can intuitively see that the program may be deadlocked in some execution.

6 Conclusion

Nondeterminism may cause deadlocks resident in multithreaded program to be hidden during testing. This paper presented a lock-causality graph for helping to analyze and identify potential deadlocks in multithreaded programs. The graph represents alternate orders over lock operations by analyzing a set of traces generated from a successful deadlock-free execution. The experiment result shows that the graph provides a simple and powerful representation of potential deadlocks in multithreaded programs not deadlocked. We have some future work to do on our graph. The first is to extend our graph for supporting different synchronization mechanisms. The second is to improve our graph for dealing with long-run and large programs.

References

1. Agarwal, A., Garg, V.K.: Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In: 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD), Portland, Maine, pp. 51–60. ACM, New York (2006)
2. Agarwal, A., Wang, L., Stoller, S.D.: Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 191–207. Springer, Heidelberg (2006)
3. Artho, C., Havelund, K., Honiden, S.: Visualization of Concurrent Program Executions. In: 31st Annual International Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 541–546. IEEE, Los Alamitos (2007)
4. Bensalem, S., Havelund, K.: Dynamic Deadlock Anaysis of Multi-threaded Programs. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 208–223. Springer, Heidelberg (2006)
5. Birrell, D.A.: An Introduction to Programming with Threads. Technical Report SR-35, Digital Equipment Corporation (January 1989)
6. Carr, S., Mayo, J., Shene, C.-K.: ThreadMentor: A Pedagogical Tool for Multithreaded Programming. Journal on Education Resources in Computing (JERIC) 3(1), 1–30 (2003)
7. Diehl, S.: Software Visualization: Visualizing the Structure, Behavior, and Evolve of Software. Springer, Heidelberg (2007)
8. Fidge, C.J.: Partial Orders for Parallel Debugging. In: SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pp. 183–194. ACM Press, New York (1988)
9. Gatlin, S.K.: Trials and Tribulations of Debugging Concurrency. Queue 2(7), 66–73 (2004)
10. Kraemer, E.: Visualizing Concurrent Programs. In: Software Visualization: Programming as a Multimedia Experience, pp. 237–258. The MIT Press, Cambridge (1998)
11. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communication of the ACM 21(7), 558–565 (1978)
12. Leroux, H., Requile-Romanczuk, A., Mingins, C.: JACOT: A Tool to Dynamically Visualise the Execution of Concurrent Java Programs. In: 2nd Int. Conf. on Principles and Practice of Programming in Java (PPPJ), Kilkenny City, Ireland, pp. 201–206. ACM, New York (2003)
13. McDowell, C.E., Helmbold, D.P.: Debugging Concurrent Programs. Computing Surveys 21(4), 593–622 (1989)
14. Mehner, K.: JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 163–175. Springer, Heidelberg (2002)
15. Sanden, B.: Coping with Java Threads. Computer 37(4), 20–27 (2004)
16. Zhao, Q.A., Stasko, J.T.: Visualizing the Execution of Threads-based Parallel Programs. Technical Report GIT-GVU-95-01, College of Computing, George Institute of Technology (January 1995)

Fragmentation of Numerical Algorithms for the Parallel Subroutines Library[*]

Victor E. Malyshkin, Sergey B. Sorokin, and Ksenia G. Chajuk

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

Abstract. Fragmentation of the often used numerical algorithms for inclusion into the library of parallel numerical subroutines are considered. Algorithms and programs fragmentation allow to create parallel programs that can be executed on parallel computers of different types (multiprocessors and/or multicomputers) and can be dynamically tuned to all the available resources. Programs' fragmentation is the way of automatic providing of the dynamic properties of parallel programs, like dynamic load balancing. Algorithm's fragmentation is a technological method of numerical algorithms parallelization which provides their effective parallel implementation.

Keywords: Asynchronous programming, parallel program, numerical algorithm, fragments based programming, dynamic programs' properties.

1 Introduction

The role that libraries of standard subroutines play in the sequential implementation of numerical models is well known. The development of similar libraries of parallel numerical subroutines is faced with a number of difficulties. The problems are caused by the necessity to provide automatically the dynamic properties of application parallel subroutines such as dynamic tunability to all the available resources of a computer in the course of execution, internodes data transfer in parallel with the program execution to reduce overhead, dynamic load balancing and the others. The organization of a library subroutines and execution of their calls from sequential and/or parallel application programs should be made in such a way in order to avoid the necessity to program dynamic properties.

For existing multicomputers many libraries of numerical subroutines were developed [1-8]. Basing on algorithm and program fragmentation, we plan to develop the portable library of subroutines for numerical modeling that should also provide automatically the dynamic properties of an application program.

[*] The work was supported by the grant Rosobrazovanie, contract RNP.2.2.1.1.3653 and by the grant of the integrating program of Siberian Branch of the Russian Academy of Science IP 5.1.

2 The Necessary Dynamic Properties of a Parallel Program

We consider that any good numerical parallel subprogram for a multicomputer should possess a number of general necessary properties.

1. In the course of execution a parallel program is represented as *a set of executed in parallel and interacting processes*[1]. The processes are assigned for execution to different nodes of multicomputer providing equal workload of all the nodes.
2. *Nondeterministic execution*. On the set of processes the partial order of the processes execution is defined in some way, for example, like this is done in asynchronous model of computations.
3. *Tunability to all the available resources of a multicomputer*. The executed program should use all the available and necessary resources of a multicomputer.
4. *Dynamic load balancing*. Depending on input data, on different stages of execution, the processes might consume substantially different volume of resources (for example, depending on number of iterations of an internal loop). In similar cases a part of processes from overloaded processor element (PE) should migrate to a neighbour underloaded node equalizing their workload.
5. *Portability among multicomputers of some class*. Portability, in particular, means, that the structure and configuration of a multicomputer, volume of its resources, should not be reflected in the code of an application parallel program.
6. *Numerical model behavioral dynamism*. Dynamic load balancing algorithms in a program, implementing large scale numerical model, depends on the numerical model behavior. In particular, prevailing direction of particles moving in the model of plasma energy exchange (implemented with Particle-In-Cell method [9]) can be used in order to construct the best plan of workload re-balancing. This substantially reduces the number of time consumed operations of the dynamic workload re-balancing.
7. Dynamic execution of the procedure call like **call** *Proc(M, ...)* where *Proc* is fragmented numerical subroutine, *M* is the distributed array. This ***call*** should not depend on array *M* distribution and/or migration of its parts (*M* distribution and/or migration should not be described in the text of a program).

3 Numerical Algorithms Fragmentation

Technology of program construction out of ready made fragments/modules is well known and the execution of such a fragmented program with the use of a run-time system was discussed in numerous publications [10-17].

Our approach to fragmented programming is based on the method of parallel program synthesis [17]. The general method of parallel programs construction is reduced to parallel programming language and system, peculiarities of numerical algorithms representation and execution are included into the language. For fragmented numerical programs execution the FLS run time system [16] was developed, that knows from compiler the structure of a numerical fragmented algorithm and

[1] *Process* – an executed program + its input/output data.

uses this knowledge for optimization of the algorithm execution. In such a way, the library of the fragmented numerical subroutines is not initially considered as a universal tool. It is especially developed as the library of subroutines, oriented to high performance implementation of the large scale numerical models of natural phenomena. This orientation is mostly embodied in the algorithms of the run-time system and the algorithms fragmentation. All the above listed dynamic properties of fragmented program are provided by the FLS. The development of the set of numerical algorithms represented in the fragmented form (fragmented algorithms) is the kernel of the library creation project.

The ideas of the numerical algorithms fragmentation are demonstrated below by a number of examples of widely used numerical algorithms fragmentation, development of the fragmented programs and their testing.

In this section the examples of fragmentation of several well known numerical algorithms are given. The description of every algorithm is done in the same style. At the beginning the initial algorithm is shortly described. Then its fragmented version is given. Finally, the performance of the sequential implementation of the initial algorithm and the parallel implementation of its fragmented version are compared. In sequential program, implementing an initial algorithm, the matrices are represented as a whole whereas in fragmented program the matrices are located as a set of submatrices. In parallel implementation of the fragmented version anywhere the sequential implementation was used, if possible. All the tested programs were developed with C language.

All test were executed on smp4x64.sscc.ru. This is four-processors server HP Integrity rx4640 with common memory. It includes 4 processors Intel Itanium2/ 1,5 GH, 4 Mb cache each, 64 GB memory with the bus bandwidth 12,8 GB/sec.

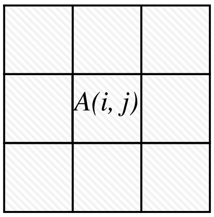
3.1 Matrices Multiplication

Initial Algorithm. The following algorithm of square matrices multiplication is used:

$$C{=}A{\times}B,\quad A=(a_{ij})_{i,j=\overline{1,N}},\quad B=(b_{ij})_{i,j=\overline{1,N}},\quad C=(c_{ij})_{i,j=\overline{1,N}},$$

$$c_{i,j}=\sum_{k=1}^{N}a_{i,l}\times b_{l,j},\quad i,j,l=1,2,\dots,N. \qquad (1)$$

This matrices multiplication is implemented by a sequential library subroutine MMul.



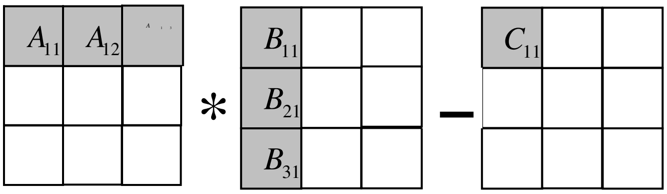


Fig. 1. Fig. 2.

Fragmented Algorithm. Matrices *A,B,* and *C* are divided into the submatrices (data fragments) of the same size (fig. 1), containing M lines and M columns. If there are $K \times K$ data fragments, then *M=M/K* and it is possible to assign a pair of indexes *I, J* , *I,J =1,…K* to every data fragment. In a Fig. 1 the case *K*=3 , *M= N/3* is presented The output data fragments $C_{I,J}$ are computed as

$$C_{I,J} = \sum_{L=1}^{K} A_{I,L} \times B_{L,J} = \sum_{L=1}^{K} C_{I,J}^{L} \tag{2}$$

where $I, J, L = 1,2,...,K$. The matrices multiplication $C_{I,J}^{L} = A_{I,L} \times B_{L,J}$ will be implemented by a sequential library procedure MMul. The formula (2) is illustrated in Fig. 3 in the case $I = J = 1$.

Therefore, in partial order to compute the output submatrix $C_{I,J}$, the program should be generated in which:

1. first, the MMul subroutine (*code fragment*) is applied to the input data fragments $A_{I,L}$ and $B_{L,J}$ in order to compute the output the data fragments $C_{I,J}^{L} = A_{I,L} \times B_{L,J}$, $I, J, L = 1,2,...,K$. The computation of the data fragments $C^{L}_{I,J}$, $L = 1,2,...,K$, can be done in arbitrary order.

A code fragment and its processed input and output data fragments are jointly called *fragment of computation*. In the course of execution the fragment of computation is called *process*.

2 when all the data fragments $C_{I,J}^{L} = A_{I,L} \times B_{L,J}$ for a certain values of *I* and *J* are computed, the sum $C_{I,J} = \sum_{L=1}^{K} C_{I,J}^{L}$ can be computed by a SUM subroutine.

In order to compute the whole matrix *C*, all the data fragments $C_{I,J}$ should be computed.

Testing. The implementation of the fragmented version is always done in such a way that the size of a data fragment is set by a user before the program execution and can be changed from one execution to another.

The sequential subroutines MMul (1) and SUM are used as a code fragments. The formed fragments of computation are included into the common queue in proper order. An idle PE choose from the common queue next fragment of computation for execution, checks the readiness of all the fragment inputs and starts its execution, until all the processes are completed.

After the completion of all the processes the result of initial matrices A and B multiplication can be found in the matrix C .

Table 1 below contains the results of testing of the fragmented program.

In columns 3, 5, 7 the execution times of fragmented subrputines are given. In columns 4, 6, 8 the speed-ups are presented. The speed-up is calculated as *a(n,k)* = $t_s/t_f(n,k)$, where t_s is the time of execution of the sequential program, and $t_f(n,k)$ is the execution time of the fragmented subroutine (*k* fragments) on *n* PE.

Table 1.

Number of fragments	Size of fragment	Execution time and speed-up of fragmented matrices 1000X1000 multiplication in comparison with the execution time of the sequential program (360,87 e-6 s)					
		1 PE	speed-up	2 PE	speed-up	4 PE	speed-up
1	2	3	4	5	6	7	8
4	500x500	356,99	1,01	180,26	2,00	63,51	5,68
16	250x250	302,36	1,19	167,54	2,15	55,79	6,47
25	200x200	***294,64***	***1,22***	159,27	2,27	55,09	6,55
100	100x100	303,77	1,19	***156,68***	***2,30***	55,22	6,54
400	50x50	306,09	1,18	163,28	2,21	***54,69***	***6,60***
1600	25x25	324,07	1,11	182,81	1,97	54,71	6,60

Resulting graphics is given in Fig. 3.

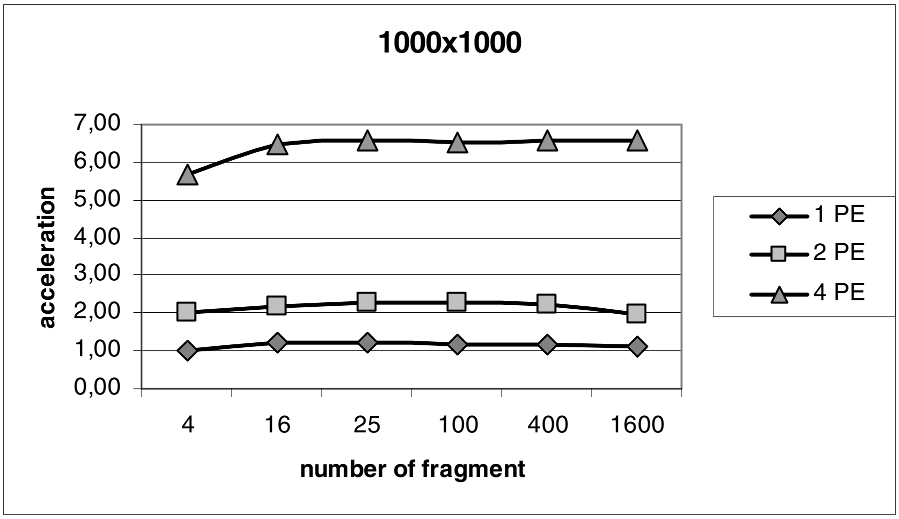


Fig. 3.

3.2 Matrix LU-Factorization

Algorithm of a matrix $A=LU$ [18] factirization into right triangular U and left triangular L matrices is:

The elements of matrices $L = (l_{ij})_{i,j=\overline{1,N}}$ and $U = (u_{ij})_{i,j=\overline{1,N}}$ are calculated recurrently following the formulae 3.

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \quad u_{ij} = \frac{1}{l_{ii}} \left[a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right]. \tag{3}$$

Data Fragments (DF): **T**he matrix *A, L,* and *U* are divided into the DFs (Fig.1).

Code Fragments (CF): The only CF is used. It calculates:

- If *i>j* then *Lij.* Input: DF*(i,1)*, …, DF*(i,j)*, DF*(1,j)*, …, DF*(i,j).* Output: DF*(i,j).*
- If *i<j* then *Uij.* Input: DF*(i,1)*, …, DF*(i,i)*, DF*(1,j)*, ..., DF*(i,j).* Output: DF*(i,j).*
- If *i=j* then *Lij* and *Uij.* Input: DF*(1,j)*, …, DF*(i,j)*, DF*(i,1)*, …, DF*(i,j).* Output: DF*(i,j).*

Computation Fragments (CompF) and the Partial Order on the Set of All CompF:

- CompF(1,1): CF is applied to DF(1,1). Input: DF(1,1). Output: DF(1,1).
- CompF(1,*n1*): For every positive integer *n1*, *n1=2..K*, CF is applied to DF(1,*n1*). Input DF(1,1), …, DF(1,*n1*). Output: DF(1,*n1*).
- CompF(*n1*,1): For every positive integer *n1*, *n1=2..K*, CF is applied to DF(*n1*,1). Input DF(1,1), …, DF(*n1*,1). Output: DF(*n1*,1).

The partial order on the set of all the CompF(1,*n1*) and CompF(*n2*,1) is not defined. All the computation fragments can be executed in arbitrary order. All these CompF are less in partial order ρ than CompF(1,1), i.e., ρ ∍ (<(1,1),(1,2)>, <(1,1),(1,3)>, …, <(1,1),(1,n1)>) ∪ (<(1,1),(2,1)>, <(1,1),(3,1)>, … , <(1,1),(*n2*,1)>).

- CompF(2,2): CF is applied to DF(2,2). Input: DF(2,1), DF(1,2), DF(2,2). Output: DF(2,2). Comp(2,2) is less in partial order ρ than CompF(1,2) and CompF(2,1).
- CompF(2,*n1*): For every positive integer *n1*, *n1=3..K*, CF is applied to DF*(2,n1).* Input DF*(2,1)*, DF*(2,2)*, DF*(1,n1)*, DF*(2,n1).* Output: DF*(2,n1).*
- CompF(*n1*,2): For every positive integer *n1*, *n1=3..K*, CF is applied to DF*(n1,2).* Input DF*(1,2)*, …, DF*(n1,2)*, DF*(n1,1)*, DF*(n1,2).* Output: DF*(n1,2).*

Order on CompF*(2,3)*, …, CompF*(2,n1)* and CompF*(3,2)*, … CompF*(n2,2)* is not defined, all fragments can be execute in arbitrary order. All these CompFs in partial order ρ less than CompF*(2,2)*, i.e., ρ*∍(<(2,2),(2,3)>,<(2,2),(2,4)>, …,<(2,2),(2,n2)>)*∪*(<(2,2),(3,2)>,<(2,2),(4,2)>,<(2,2),(n1,2)>).* And go on.

After the CompF*(K,K)* execution is completed the matrices *L* and *U* are computed (Fig. 4).

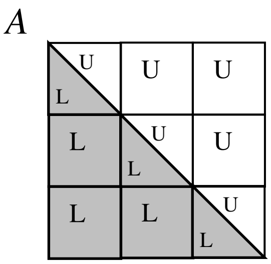


Fig. 4.

The defined order is shown in a Fig. 5.

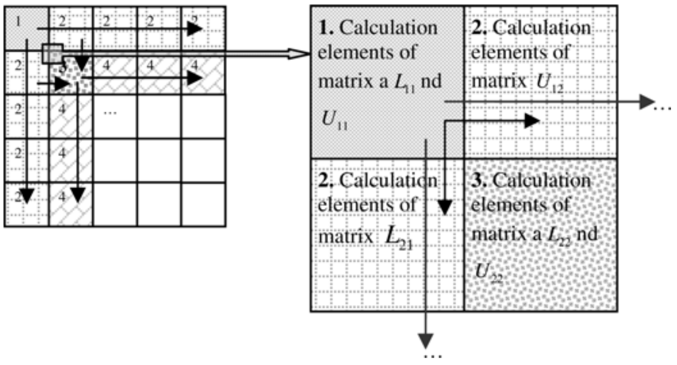

1
2
2
2
2
2
2
2
2
3
4
4
4
4
4
4
...
1. Calculation elements of matrix a L_{11} nd U_{11}
2. Calculation elements of matrix U_{12}
2. Calculation elements of matrix L_{21}
3. Calculation elements of matrix a L_{22} nd U_{22}
...
...

Fig. 5.

Table 2.

Number of fragments	Size of fragment	Execution time and speed-up of fragmented matrices 10000X10000 factorization in comparison with the execution time of the sequential program (3.54 s)					
		1 PE	Speed-up	2 PE	Speed-up	4 PE	Speed-up
4	5000	3,76	0,94	1,93	1,83	1,13	3,13
16	2500	3,61	0,98	1,84	1,92	1,09	3,25
100	1000	***3,52***	***1,01***	1,81	1,96	0,98	3,61
400	500	3,55	1,00	***1,73***	***2,05***	0,91	3,89
1600	250	3,59	0,99	1,76	2,01	***0,87***	***4,07***
2500	200	3,67	0,96	1,82	1,95	0,90	3,93
10000	100	3,76	0,94	1,85	1,91	0,97	3,65
40000	50	3,84	0,92	1,91	1,85	1,06	3,34

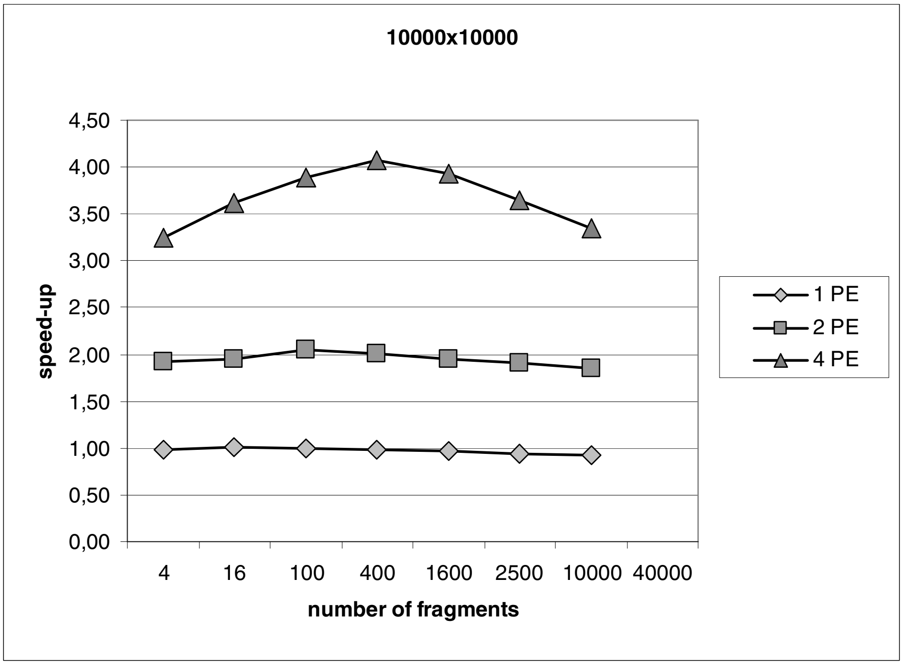


Fig. 6.

Testing. The result of testing are presented in Table 2 and Fig. 6.

3.3 QR-Factorization

QR-factorization [18] of a matrix *A, N*×*N,* is calculated by multiplication $R=A\times Q_1\times\ldots\times Q_N$, where Q_i are orthogonal transformations of rotation.

Multiplication $A\times Q_1$ will zero all the elements of a matrix *A*, located in the first column of the matrix *A* under element a_{11}. Multiplication $A\times Q_1\times Q_2$ will zero all the elements of a matrix *A*, located in the second column of the matrix A under element a_{22} and go on. As result the matrix *A* will be transformed into the right triangular matrix *R*.

Example for the case of 3×3 matrix *A* is below:

$$\begin{array}{ccccccc}
\begin{array}{c}\textit{Initial}\\ \textit{matrix } A = A_1\end{array} & & A_2 = A_1\cdot Q_1 & & A_3 = A_2\cdot Q_2 & & R = A_3\cdot Q_3 \\
\begin{pmatrix} a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\\ a_{31} & a_{32} & a_{33}\end{pmatrix} & \to & \begin{pmatrix} b_{11} & b_{12} & b_{13}\\ 0 & b_{22} & b_{23}\\ 0 & b_{32} & b_{33}\end{pmatrix} & \to & \begin{pmatrix} b_{11} & b_{12} & b_{13}\\ 0 & c_{22} & c_{23}\\ 0 & 0 & c_{33}\end{pmatrix} & \to & \begin{pmatrix} b_{11} & b_{12} & b_{13}\\ 0 & c_{22} & c_{23}\\ 0 & 0 & d_{33}\end{pmatrix}
\end{array}$$

Data Fragments. The matrices A_i are divided into the DFs just as this is done in fig. 1. Intermediate matrices Q_i, j=1,2,…,K, are fragmented likewise. The same memory is used in order to keep A_i and Q_i. Actually, at any moment only column of Q_i data fragments is used and kept. Let $A_i(i,j)$ is *(i-th,j-th)* DF of A_i, $Q_i(j)$ is the DF of Q_i, where $i=1..K$, $j=1..K$.

Code Fragments . Two types of code fragments:

- CF1 - calculation of rotation matrices Q_i and the elements of matrices A_i re-calculation. Input: DFs $A_i(i,i)$ and $A_i(j,i)$. Output: $A_{i+1}(j,i)$ and $Q_i(j)$, where $i=1..K$, $j=i..K$.
- CF2 – re-calculation of matrices A_i elements from matrix Q_i. Input: DFs $A_i(j,j1)$ and $Q_i(j)$. Output: $A_{i+1}(j,j1)$, where $i=1..K$, $j=i..K$, $j1=i+1$.

Computation Fragments and the Partial Order on the Set of All CompF
QR-decomposition of a matrix A is calculated iteratively.

1. The first iteration, $i=1$.
   a) the fragment of computation CompF*(1,1)*: CF1 is applied to $A_i(1,1)$ and transform $A_i(1,1)$ into the upper triangular form (fig.7). Input: $A_i(1,1)$. Output: $A_i(1,1)$, $Q_1(1)$.

The fragments of computation CompF*(1,n1)*: For every positive integer $n1$, $n1=2,...,K$, CF2 is applied to $A_i(1,n1)$, Input: $A_i(1,n1)$ and $Q_1(1)$,. Output: $A_i(1,n1)$.

b) The fragment of computation CompF*(2,1)*: CF1 is applied to $A_i(2,1)$. Input: $A_i(1,1)$, $A(2,1)$. Output: $A_i(2,1)$, $Q_1(2)$.

The fragments of computation CompF*(2,n1)*: For every positive integer $n1$, $n1=2..K$, CF2 is applied to $A_i(2,n1)$. Input: $A_i(2,n1)$ and $Q_1(2)$. Output: $A_i(2,n1)$.

c) A fragment of computions CompF*(3,1)*: CF1 is applied to $A_i(3,1)$. Input: $A_i(1,1)$, $A_i(3,1)$. Output: $A_i(3,1)$, $Q_i(3)$.

The fragments of computation CompF*(3,n1)*: For every positive integer $n1$, $n1=2..K$, CF2 is applied to $A_i(3,n1)$. Input: $A_i(3,n1)$ and $Q_i(3)$. Output: $A_i(3,n1)$.

And so on, while all the $A_i(j,1)$ are set into zero. The other CompF*(K,n1)*, $j=2..K$, $n1=2..K$, also should be executed.

A partial order of execution on the set of computation fragments CompF*(i,j)*, $i>0$, $j=2..K$, and CompF*(i+1,1)* is not defined. Therefore, all these computation fragments can be executed in arbitrary order. These fragments of computations are less in partial order ρ than CompF*(i,1)*. Thus, partial order ρ contains the elements

$\rho \ni (<(i,1), (i,j)>, <(i,1), (i+1,1)>)$, where $i=1..K$, $j=2..K$.

The partial order ρ defines here the order of computation fragments execution inside the iteration. Iterative execution can be defined by the control loop like *for* and *while*. More complex control can be defined, for example, by Petri net.

2) The second iteration, $i:=i+1$, $i\leq K$. Similarly the computation fragments are defined for all the $A_i(i,j)$, $i=2..K$, $j=2..K$. The code fragment CF1 is applied to all the first column $A_i(i,j)$, $j=i..K$. The CF2 is applied to all the $A_i(i,j)$, $i=2..K$, $j=3..K$.
   And go on until CompF*(K,K)* is executed and matrix R is calculated.

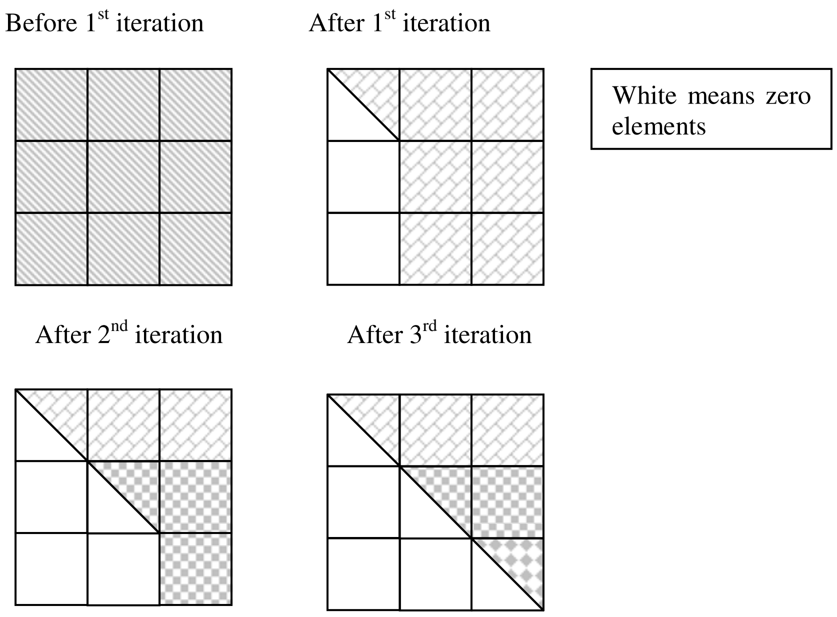


Fig.7.

Testing. The graphics below demonstrates the results of fragmented program testing (Fig. 8).

Far more complex example of Particle-In-Cell method algorithms fragmentation can be found in [9].

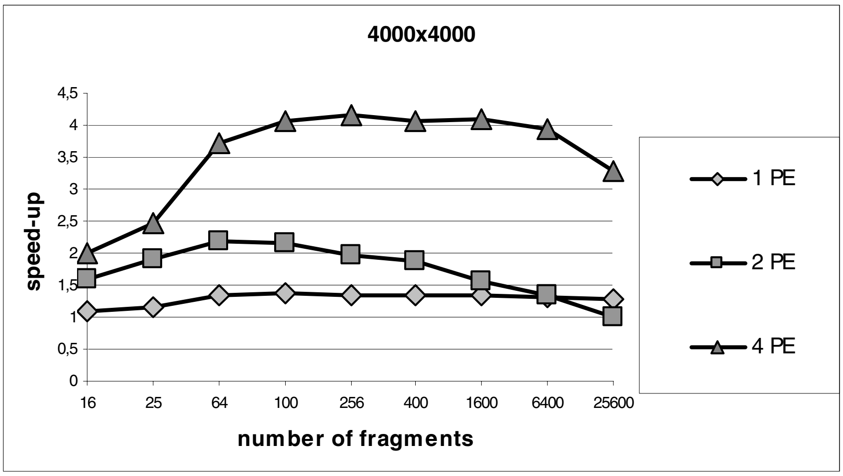


Fig. 8.

4 Qualitative Characteristics of a Fragmented Program Execution

The qualitative graphics of fragmented program execution (the same size of a problem and the same number of computer resources used) is shown in Fig. 9. The graphics can be explained by the affect of several factors.

1. There is clearly visible minimum of the total time of a fragmented program execution. Initially, the time of the program execution is decreasing. With growing the fragments number, more computer resources are involved in computation, some fragments can be executed in parallel. Then, after minimum, the time of execution is increasing, because the time of communications and control implementation began to exceed the benefits of parallel fragments execution.
2. The total time of monoprocessor execution of a fragmented program is less then the time of execution of the sequential program because of the reduction of the data access time.

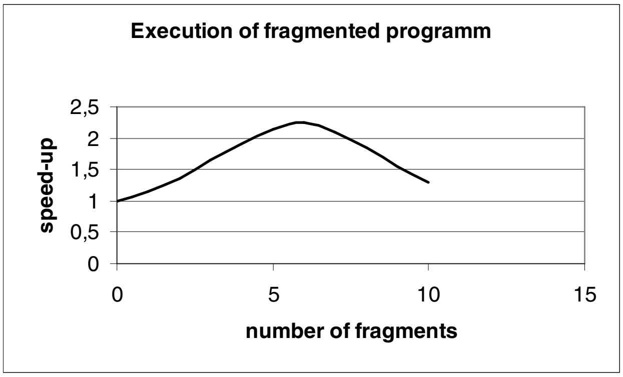


Fig. 9.

3. If the size of data fragments is decreasing (accordingly, if the number of data fragments is increasing), the fragmented program is executed faster since the data fragments begin fully to fit in cache-memory.
4. The minimal time of execution point corresponds to an optimum ratio between code fragment, control and communication execution. In this point idle time of computer resources is reduced to a minimum, size of a fragment is optimal for available resources, i.e. data fit in a cache, streams do not hinder each other, execution time allows to make all necessary communications in parallel with fragments execution.

5 Conclusion and Future Work

We develop the library of fragmented numerical subroutines where subroutines portability providing is concentrated in system software. Text of a subroutine, written in fragmented algorithm programming language (FAPL), contains a numerical algorithm description and user's not obligatory recommendations on how to execute the subroutine on parallel computer only. Actually, subroutine text contains the description of the algorithm of CompFs and processes creation.

Intelligent FAPL compiler also can collect the info on the structure of the set of CompF basing on the regular data structures of numerical algorithms and can provide runtime system by this knowledge, which is used for construction of the way of fragmented algorithm execution.

Because the performance of the subroutine execution depends mainly on the quality of compiler and run time system implementation, their improvement will be the mainstream of the project. We hope to reach good performance of system software in order even sequential program with subroutine calls could demonstrate highly effective parallel solution of numerical problems.

References

1. Glushkov, V.M., Ignatyev, M.V., Myasnikov, V.A., Torgashev, V.A.: Recursive machines and computing technologies. In: Proceedings of the IFIP Congress, vol. 1, pp. 65–70. North-Holland Publish. Co., Amsterdam (1974)
2. Hill, J., McColl, W., Stefanescu, D., Goudreau, M., Lang, K., Rao, S., Suel, T., Tsantilas, Th., Bisseling, R.: BSPlib: the BSP Programming Library. Parallel Computing 24, 1947–1980 (1998)
3. Torgashev, V.A., Tsarev, I.V.: Sredstva organozatsii parallelnykh vychislenii i programmirovaniya v multiprocessorakh s dynamicheskoi architechturoi. Programmirovanie (4), 53–67 (2001)
4. BSPlib, `http://www.bsp-worldwide.org/`
5. BLAS, `http://www.netlib.org/blas/`
6. ScaLAPACK, `http://www.netlib.org/scalapack/`
7. `http://www.intel.com/cd/software/products/emea/rus/358876.htm`
8. Charm++, `http://charm.cs.uiuc.edu/manuals/html/converse/manual.html`
9. Kraeva, M.A., Malyshkin, V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. The Int. Journal on Future Generation Computer Systems 17(6), 755–765 (2001)
10. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. ACM SIGPLAN Notices 30(8), 207–216 (1995)
11. Foster, I., Kesselman, C., Tuecke, S.: Nexus: Runtime Support for Task-Parallel Programming Languages. Cluster Computing 1(1), 95–107 (1998)
12. Shu, W., Kale, L.V.: Chare Kernel – a Runtime Support System for Parallel Computations. Journal of Parallel and Distributed Computing 11(3), 198–211 (1991)
13. Chien, A.A., Karamcheti, V., Plevyak, J.: The Concert System – Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs. - UIUC DCS Tech Report R-93-1815 (1993)
14. Grimshaw, A.S., Weissman, J.B., Strayer, W.T.: Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. ACM Transactions on Computer Systems (TOCS) 14(2), 139–170 (1996)

15. Benson, G.D., Olsson, R.A.: A Portable Run-Time System for the SR Concurrent Programming Language. In: Proceedings of the Workshop on Run-Time Systems for Parallel Programming (RTSPP) (April 1997)
16. Kalgin, K.V., Malyskin, V.E., Nechaev, S.P., Tschukin, G.A.: Runtime System for Parallel Execution of Fragmented Subroutines. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 544–552. Springer, Heidelberg (2007)
17. Valkovskii, V., Malyshkin, V.: Parallel Program Synthesis on the Basis of Computational Models. – Novosibirsk, Nauka, 129 p. (In Russian/ Sintez Parallel'nykh program ya Vychislitel'nykh modelyakh) (1988)
18. Faddeev, D.K., Faddeeva, V.N.: Computing methods of linear algebra. 2nd edn. Moscow, Nauka, p. 656 (In Russian/ Vychislitel'nye metody lineinoi algebry) (1967)

Object-Oriented Parallel Image Processing Library*

Evgeny V. Rusin

Institute of Computational Mathematics and Mathematical Geophysics SB RAS
prospect Akademika Lavrentjeva, 6, Novosibirsk, 630090, Russia
rev@ooi.sscc.ru

Abstract. The paper describes the experimental library SSCC_PIPL for image processing on multicomputers. Basic principles of library building, some architectural solutions, and test results are given.

Keywords: image processing, parallel processing, library of subprograms.

1 Introduction

Remote sensing data processing tasks are characterized by the huge amount of data to process (10^8 multispectral pixels is a typical case) and high labor-intensiveness of processing algorithms (can easily exceed 10^8 operations per pixel). The need of remote sensing data real-time analysis and interpretation (for example in flood or forest fire monitoring) causes the necessity of using high-performance computational tools. The most common type of such tools is multicomputers, MIMD systems with distributed memory.

The paper describes an experimental library SSCC_PIPL for image processing on multicomputers. The library is created in the Institute of Computational Mathematics and Mathematical Geophysics SB RAS and is intended to provide software support for high-performance remote sensing data processing.

2 State-of-Art

World experience in creating parallel image processing systems [1-6] indicates that:

1. Such libraries should guard their users from implementation details including multicomputer architecture, parallel programming system, and so on. Ideally library interface should be designed in such a way that parallel program using the library would look like the sequential one.
2. Image processing algorithms can be divided into several large groups according to their program implementation; implementations of different algorithms of the same group contain essential common part of source code. These are: a) pixel-to-pixel operations, when the result of an operation in a pixel depends only on the value of original images in the same pixel; b) neighborhood-to-pixel operations when the result of an operation in a pixel depends on the values of original images in some

* Supported by Russian Foundation for Basic Research (project No. 07-07-00085a).

relatively small neighborhood of the pixel; c) global operations when the result of an operation in a pixel depends on whole original images.

3. Most image processing algorithms are naturally parallel and the number of approaches to parallelize computations is small enough: cutting images without overlaps (for pixel-to-pixel operations), cutting with margin overlaps (for neighborhood-to-pixel operations), and cloning original images to all the executing processors (for global operations; here each processor calculates its own part of the result).
4. Permanent appearance of new algorithms requires extensibility of the systems and the simplicity of user's algorithms addition to them.

3 Basic Principles

We formulated the following principles for SSCC_PIPL library building:

1. The use of SPMD (Single Program Multiple Data) model. Parallel program using the library is compiled into executable module, and each node of multicomputer executes a copy of the module.
2. All the code responsible for parallel environment operations (initialization/termination of the environment, self-identification of a node among all the executing nodes, interprocess transfer and synchronization) is located in the library implementation. At the same time, we consider that the total hiding of parallelism from user would lead to potential ineffectiveness of program systems based on algorithm composition (one parallelization approach is most suitable for algorithm *A*; another, for algorithm *B*; and third, possibly coinciding with the first or the second, for the composition of *A* and *B*). Library user is supposed to be able to make some general considerations on effective parallelization of the algorithm (for example, optimal parallelization of the composition of three filterings with 5 by 5 window is cutting image with margin overlap in 2 (overlap required for one transformation) $\times$ 3 (number of transformations) = 6 pixels), and the library should allow user to specify parallelization approach 'in general'.
3. To avoid source code duplicating, the library should have a compact core containing several generic functions, each implements common code for the particular algorithm group. Concrete algorithm is implemented by the 'algorithm' function passed to generic function as a parameter and executing, for example, calculation of the result in one pixel. Extensibility of the library is reached by possibility to add algorithm functions. To extend the library, one does not need to deal with parallelism models but has to implement the operation in terms of image processing.
4. The library must minimize overheads caused by the abstraction level of computation model (in particular caused by using generic operation implemented in the core once and parameterized with user's code – see the previous item).

The library should be implemented in C++ and MPI. It provides the possibility of low-level optimization, portability, as well as generalization with mechanisms of inheritance and templates. Besides, MPI corresponds to SPMD model, and the support of object-oriented paradigm by C++ language provides the high degree of encapsulation of the implementation details behind high-level interface.

4 Library Interface

The following classes constitute SSCC_PIPL library interface:

- `RunTime`, run-time environment. Provides auxiliary operations (MPI initialization/termination, debug messaging, time gaps measuring, and so on).
- `Image`, class implementing image abstraction. Provides methods for reading/writing image from/to file in various graphic formats and transforming images.
- `NeighborhoodManipulator`, manipulator of image pixel neighborhood. Allows implementing processing algorithms throwing off concrete image parameters (size, distribution among processors, and so on) and thinking only in the terms of neighborhood of the pixel being processed. The use of neighborhood manipulator abstraction simplifies the creation of library-compatible user algorithm functions.

5 Parallelization of Algorithms

The library provides the following ways to distribute an image between processors:

- Full copy of the image on each processor.
- Cutting the image onto non-overlapping horizontal strips whose number equals the number of processors; each processor receives its own strip to process.
- Cutting the image onto overlapping horizontal strips: neighbor strips overlap in a given number of rows; each processor receives its own strip and strip margin data are duplicated on two processors. `Image` object maintains the validity of data in the overlap areas and this maintenance is transparent for user.

6 Input-Output Operations

SSCC_PIPL library provides reading and writing images in various graphic formats by using CxImage library [7]. CxImage is ANSI-compatible and can be used in UNIX, Windows and MacOS environments. It is freely distributed in source code and does not require additional licensing. However CxImage is sequential library and does not support ‘distributedness’ of an image among several computers. That’s why SSCC_PIPL executes input-output operations using star-like topology: one executing processor is marked as a root and performs disk operations. Thus, reading image from file is performed by ‘the root reads file with CxImage and sends necessary data to other processors’ scheme, and writing image to file is performed by ‘the root gathers whole image data from other processors and writes it to file with CxImage’ scheme.

7 Parametrization of Operations with Algorithms

Image transformations are implemented in SSCC_PIPL library as generic operations. To apply concrete transformation to an image, one should parametrize the corresponding generic operation by concrete algorithm. At the design stage, the choice appeared between two C++ parametrization mechanisms:

- Inheritance: generic operation argument is reference to abstract base class (interface) with the pure virtual `Process()` method which performs calculation of new value in a pixel by the current pixel values; concrete algorithm classes are derived from the base and overrides `Process()` method.
- Templates: generic operation is a template method parametrized be concrete algorithm class.

From considerations of performance and universality, SSCC_PIPL uses the second approach; this allowed carrying overheads caused by the increasing of computation model abstraction level from program execution time to compile time. Besides, such an approach allows so-called embedding of the function that calculates pixel value that is inserting a copy of the function body into each place the function is called. Embedding is not available for virtual functions. As a result of this choice, the library must be distributed in source code, user's program compiles longer, but executes faster.

8 Sample Source Code

The following simplified program illustrates typical use case of the library, defining user algorithm and parametrizing corresponding `Image` template method with it.

```
1.   class Laplasian {
2.      int LeftMargin() { return 1; }
3.      int RightMargin() { return 1; }
4.      int TopMargin() { return 1; }
5.      int BottomMargin(){ return 1; }
6.      int Process(NeighborhoodManipulator& nm) {
7.         return (nm.PRC(-1, 0) + nm.PRC(0, 1)
8.                + nm.PRC(1, 0) + nm.PRC(0, -1))/4
9.                   - nm.PRC(0, 0);
10.     }
11.  };
12.  void main(int argc, char* argv[]) {
13.     Image im;
14.     Partitioning p(CutWithOverlap, 1);
15.     im.Create("<SOURCE_IMAGE>", pi);
16.     Laplasian l;
17.     im.N2P<Laplasian>(l);
18.     im.Save("<DEST_IMAGE>");
19.  }
```

`Laplacian` class (lines 1-11) implements discrete Laplace transformation. Laplace transformation of the image $X = \{x_{i,j}\}$ is defined as the image $Y = \{y_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4 - x_{i,j}\}$. This is a transformation of neighborhood-to-pixel type. `Laplacian` object is passed as a parameter to `Image::N2P()` ('neighborhood-to-pixel' operation) template method in line 17. Image transformation class must implement:

1. Limiters (lines 2-5) that define the minimal square window needed to perform calculations in a single pixel. For example, if `LeftMargin()` returns k, then processing of pixel (i, j) involves pixels from $j - 1, j - 2, \ldots, j - k$ columns. Limiters allow the environment to avoid calculations in image margin pixels and to decide whether processors have enough information to perform operation in strip margin pixels (if not, additional pixel values will be taken from the neighbor processors). In the example above, processing requires 3 by 3 window centered at the pixel being processed, and all the limiters return 1.
2. `Process()` method (lines 6-10) that performs calculation in single image pixel. The method accepts `NeighborhoodManipulator` object which allow referencing pixels around the one being processed via `PRC()` ('pixel relative to current') method. Environment calls this method for each image pixel.

Line 14 declares the partitioning object describing how to distribute the image among processors. Here we cut the image into horizontal strips with one-pixel overlap; this is enough to perform Laplace transformation without interprocessor communications.

Line 15 loads the image from the disk file and distributes it between processors.

Line 17 executes image parallel transformation: environment determines (by using transformation's limiters) whether executing processors have enough information for calculation in strip margin pixels; initiates interprocessor exchange if there is no enough information; and calls transformation's `Process()` method for each pixel.

Line 18 stores the result image in the disk file.

As one can see, the code looks like a usual sequential program; all parallel code is hidden in `Image` class implementation. Library user can implement their own image transformation class and use it as a parameter for `Image::N2P()` or other template method, setting desired parallelizing technique in `Partitioning` object.

9 Experimental Results

Answers on two main questions that are necessary to justify the chosen approach were obtained in a result of test calculations.

The first question: Is overhead caused by the computation model abstraction level reasonable? In order to answer the question, two parallel programs implementing the algorithm of circle structure detection in aerospace images [8] were created, one was written with SSCC_PIPL library (P_1), and other was written 'from nothing' and explicitly called MPI subroutines, distributed data among processors, dealt with synchronization, and so on (P_2). The results are:

1. Writing P_1 took much less time than writing P_2.
2. The source code of P_1 is much more compact than the one of P_2.
3. P_2 works approximately 10 percents faster than P_1.

Thus we can state that the suggested approach substantially speeds up and simplifies the development of parallel image processing programs at the expense of small decrease of program performance.

The second question: What parallelization efficiency does the library provide? Tests executed on MVS-1000/M multicomputer of Siberian Supercomputer Center showed that parallelization efficiency of circle structure detection algorithm on 16 processors is about 95 percents, and this allows saying about perspectiveness of the suggested approach.

References

1. Bräunl, T.: Tutorial in Data Parallel Image Processing. Australian Journal of Intelligent Information Processing Systems 6(3), 164–174 (2001)
2. Jamieson, L.H., Delp, E.J., Patel, J.N., Wang, C.C.: A Library-based Program Development Environment for Parallel Image Processing. In: Scalable Parallel Libraries Conference, pp. 187–194 (1993)
3. Lebak, J., Kepner, J., Hoffmann, H., Rutledge, E.: Parallel VSIPL++: An Open Standard Software Library for High-Performance Parallel Signal Processing. Proc. IEEE 93(2), 313–330 (2005)
4. Seinstra, F.J., Koelma, D.: The Lazy Programmer's Approach to Building a Parallel Image Processing Library. In: Parallel and Distributed Processing Symposium, pp. 1169–1176 (2001)
5. Seinstra, F.J., Koelma, D., Geusebroek, J.M., Verster, F.C., Smeulders, A.W.M.: Efficient applications in user transparent parallel image processing. In: Parallel and Distributed Processing Symposium, pp. 127–134 (2002)
6. Squyres, J.M., Lumsdaine, A., Stevenson, R.L.: A Toolkit for Parallel Image Processing. In: SPIE Proc., vol. 3452, pp. 69–80 (1998)
7. CxImage, `http://www.xdp.it/cximage.htm`
8. Alekseev, A.S., Pyatkin, V.P., Salov, G.I.: Crater Detection in Aero-Space Imagery Using Simple Nonparametric Statistical Tests. In: Chetverikov, D., Kropatsch, W.G. (eds.) CAIP 1993. LNCS, vol. 719, pp. 793–799. Springer, Heidelberg (1993)

Application-Level and Job-Flow Scheduling: An Approach for Achieving Quality of Service in Distributed Computing

Victor Toporkov

Computer Science Department, Moscow Power Engineering Institute,
ul. Krasnokazarmennaya 14, Moscow, 111250 Russia
ToporkovVV@mpei.ru

Abstract. This paper presents the scheduling strategies framework for distributed computing. The fact that architecture of the computational environment is distributed, heterogeneous, and dynamic along with autonomy of processor nodes, makes it much more difficult to manage and assign resources for job execution which fulfils user expectations for quality of service (QoS). The strategies are implemented using a combination of job-flow and application-level techniques of scheduling and resource co-allocation within virtual organizations of Grid. Applications are regarded as compound jobs with a complex structure containing several tasks. Strategy is considered as a set of possible job scheduling variants with a coordinated allocation of the tasks to the processor nodes. The choice of the specific variant depends on the load level of the resource dynamics and is formed as a resource request, which is sent to a local batch-job management system.

Keywords: scheduling, resource co-allocation, strategy, job, task, critical work.

1 Introduction

The fact that a distributed computational environment is heterogeneous and dynamic along with the autonomy of processor nodes makes it much more difficult to manage and assign resources for job execution at the required quality level [1]. Job management issues including resource allocation and scheduling are addressed by a number of research groups. Dealing with the wide range of different approaches to distributed computing, one can pick out two polar and settled trends. First one is based on the usage of the available and non-dedicated resources, where resource brokers are acting as agents between users and processor nodes [2-4]. Several projects such as AppLeS [5], APST [6], Legion [7], DRM [8], Condor-G [9], Nimrod/G [10] and others, which follow this idea, are often associated with application-level scheduling. Another trend is based on the concept of virtual organizations and is mainly aimed at Grid systems [1, 11, 12]. Both trends have their own advantages and disadvantages.

Resource brokers, that are used in the first trend [2-10] are scalable and flexible and can be adapted to a specific application. However resource distribution dedicated to the application-level as well as the usage of different criteria by independent users for the respective job scheduling optimization [12], while considering possible competition with other jobs, may deteriorate such integral characteristics as completion time for the batch-job or resource load level [4, 13].

Forming virtual organizations [1] is essentially bordering the scalability of the scheduling framework, although the set of the specific rules for job-flow assignment and resource consuming [14] allows an overall increase in the efficiency of batch-job scheduling and resource usage. Completion time for single jobs can be longer, because the structures of the jobs for some user-specific needs are not taken into account during the scheduling. In order to control the flow of independent jobs, special metaschedulers, managers, Grid-dispatchers [11, 15] are acting as agents between users and local batch-job systems. One can mention that the alternative Grid resource structure has a single central entity, that is controlling the flow of all jobs and no local schedulers are used (commercial platforms DCGrid, LiveCluster, GridMP, Frontier, volunteer projects @Home [16] and CCS system, which supports dedicated resources).

Distinct from existing Grid scheduling solutions [2-16], our approach supposes techniques of dynamic redistribution of job-flows between processor nodes in conjunction with application-level scheduling. It is considered, that the job can be compound (multiprocessor) and the tasks, included in the job, are heterogeneous in terms of computation volume and resource need. In order to complete the job, one would co-allocate [17] the tasks to different nodes. Each task is executed on a single node and it is supposed, that the local management system interprets it as a job accompanied by a resource request.

On one hand, the structure of the job is usually not taken into account [13]. The rare exception is the Maui cluster scheduler, which allows for a single job to contain several parallel, but homogeneous (in terms of resource requirements) tasks. On the other hand, there are several resource-query languages. Thus, JDL from WLMS defines alternatives and preferences when making resource query, ClassAds extensions in Condor-G [9] allows forming resource-queries for dependant jobs. The execution of compound jobs is also supported by WLMS scheduling system of gLite platform, though the resource requirements of specific components are not taken into account.

What sets our work apart from other scheduling research is that we consider coordinated application-level and job-flow management as a fundamental part of the effective scheduling strategy within the virtual organization. The choice of the strategy depends on the utilization state of processor nodes [4, 13], data storage and replication policies [11, 18, 19], the job structure (computational granularity and data dependencies), user estimations of completion time, resource requirements, and advance reservations [20].

The outline of the paper is as follows. Section 2 presents a framework for integrated job-flow and application-level scheduling. In section 3, we provide details of our approach based on strategies as sets of possible supporting schedules. Simulation studies of coordinated scheduling techniques and results are discussed in Section 4. We conclude and point to future directions in Section 5.

2 Scheduling Framework

In order to implement the effective coordinated scheduling [17] and allocation to heterogeneous resources [13], it is very important to group user jobs into flows according to the strategy selected. A hierarchical structure (Fig. 1) composed of a job-flow metascheduler and subsidiary job managers, which are cooperating with local batch-job management systems, is a core part of a scheduling framework proposed in this paper. The advantages of hierarchically organized resources managers are obvious, e.g., the hierarchical job-queue-control model is used in the GrADS metascheduler [15]. Hierarchy of intermediate servers allows decreasing idle time for the processor nodes, which can be inflicted by transport delays or by unavailability of the managing server while it is dealing with the other processor nodes. Tree-view manager structure in the network environment of distributed computing allows avoiding deadlocks when accessing resources. Another important aspect of computing in heterogeneous environments is that processor nodes with the similar architecture, contents, administrating policy are grouped together under the node manager control.

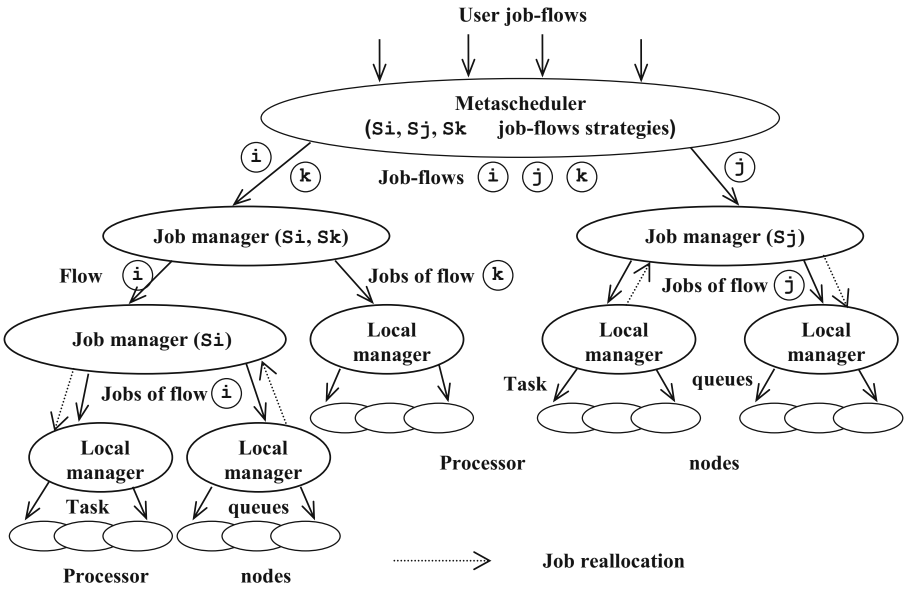


Fig. 1. Hierarchical structure of the scheduling framework

Users submit jobs to the metascheduler (see Fig. 1) which distributes job-flows between processor node domains according to the selected scheduling and resource co-allocation strategy Si, Sj or Sk. It does not mean, that these flows cannot "intersect" each other on nodes. The special reallocation mechanism is provided. It is executed on the higher-level manager or on the metascheduler-level. Job managers are supporting and updating strategies based on cooperation with local managers and simulation approach for job execution on processor nodes.

Innovation of our approach consists in mechanisms of dynamic job-flow environment reallocation based on scheduling strategies. The nature of distributed computational environments itself demands the development of multicriteria [21] and multifactor [22] strategies of coordinated scheduling and resource allocation. The dynamic configuration of the environment, large number of resource reallocation events, user's and resource owner's needs as well as virtual organization policy of resource assignment should be taken into account. The scheduling strategy is formed on a basis of formalized efficiency criteria, which sufficiently allow reflecting economical principles [14] of resource allocation by using relevant cost functions and solving the load balance problem for heterogeneous processor nodes. The strategy is built by using methods of dynamic programming [23] in a way that allows optimizing scheduling and resource allocation for a set of tasks, comprising the compound job.

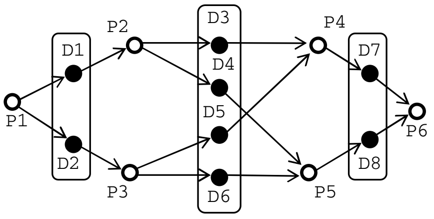
3 Scheduling Strategies

The strategy is a set of possible resource allocation and schedules (distributions) for all `N` tasks in the job:

```
Distribution:=<<Task 1/Allocation i,[Start 1, End 1]>,
      …, <Task N/Allocation j, [Start N, End N]>>,
```

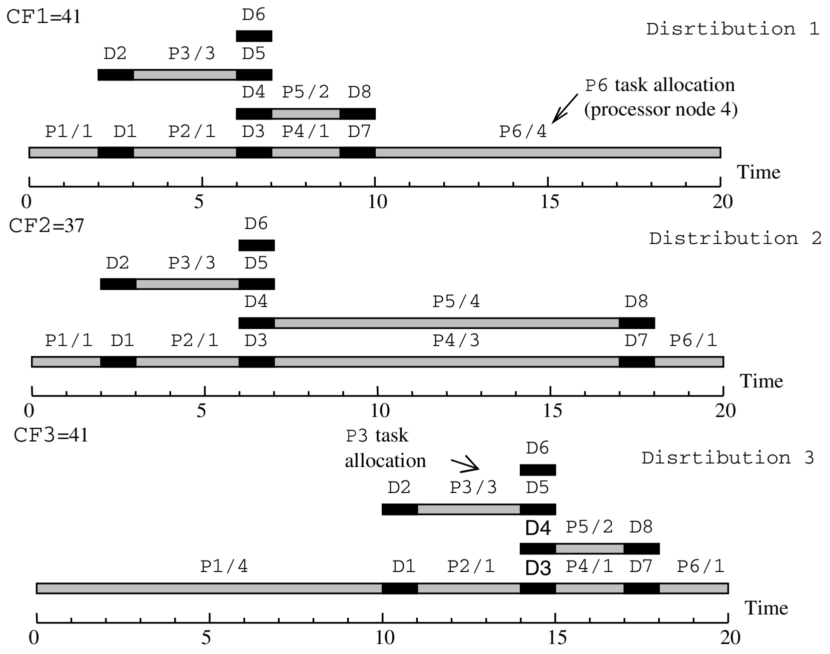
where `Allocation i, j` is the processor node `i, j` for `Task 1, N`; `Start 1, N`, `End 1, N` – run time and stop time for `Task 1, N` execution. Time interval `[Start, End]` is treated as so called wall time, defined at the resource reservation time [20] in the local batch-job management system.

Figure 2 shows an exemplary information graph of a compound job with user task estimations (Fig. 2, a) and a fragment of the strategy with `Distribution` variants for schedules and co-allocations (Fig. 2, b). Vertices `P1`, ..., `P6` are corresponding to tasks, `D1`, ... , `D8` – to data transfers. `Distribution 2` (see Fig. 2, b) provides minimum of a job execution cost-function `CF2=37` equal to the sum of `Vij/Ti`, `i=1,…,N`, where `Vij` is the relative computation volume, and `Ti` is the real load time of processor node `j` by task `i` (rounded to nearest not-smaller integer). Obviously, actual solving time `Ti` for a task can be different from user estimation `Tij`. It is to mention, such estimations are also necessary in several methods of priority scheduling including backfilling in Maui cluster scheduler. Cost-functions can be used in economical models [14] of resource distribution in virtual organizations and it is worth noting that full costing in `CF` is not calculated in real money, but in some conventional units (quotas), for example like in corporate non-commercial virtual organizations. The essential point is different – user should pay additional cost in order to use more powerful resource or to start the task faster. For that reason `Distributions 1, 3` for the job in general (see Fig. 2, b) "cost" more (`CF1=CF3=41`). The choice of a specific `Distribution` from the strategy depends on the state and load level of processor nodes, and data storage policies.



Task estimations	Tasks					
	P1	P2	P3	P4	P5	P6
Ti1	2	3	1	2	1	2
Ti2	4	6	2	4	2	4
Ti3	6	9	3	6	3	6
Ti4	8	12	4	8	4	8
Vij	20	30	10	20	10	20

(a)



(b)

Fig. 2. Job graph with user's estimations (a) and the fragment of the scheduling strategy (b)

A critical works method [23], which was developed for application-level scheduling, can be further refined to build multifactor and multicriteria strategies for job-flow distribution in virtual organizations. This method is based on dynamic programming and therefore uses some integral characteristics, for example total resource usage cost for the tasks that compose the job. However the method of critical works can be referred to the priority scheduling class. There is no conflict between these two facts, because the method is dedicated for task co-allocation of compound jobs.

The gist of the method is a multiphase procedure, which is searching for a next critical work – the longest (in terms of estimated execution time) chain of unassigned tasks along with the best combination of available resources, and resolving collisions cased by conflicts between tasks of different critical works competing for the same resource. As shown on Fig. 2, a, there are four critical works 12, 11, 10, and 9 time units long (including data transfer time) on fastest processor nodes of the type `1`:

`P1-P2-P4-P6, P1-P2-P5-P6, P1-P3-P4-P6, P1-P3-P5-P6.`

`Distribution 2` has a collision (see Fig. 2, b), which occurred due to simultaneous attempts of tasks `P4` and `P5` to occupy processor node `3`. This collision is further resolved by the allocation of `P4` to the processor node `3` and `P5` to the node `4`. Such reallocations can be based on virtual organization economics – in order to take higher performance processor node, user should "pay" more. The main positions of the critical works method are described in earlier papers [21-23].

4 Simulations Studies and Results

We have implemented a simulation environment of the scheduling framework to evaluate efficiency indices of different scheduling and co-allocation strategies. In contrast to well-known Grid simulation systems such as ChicSim [11] or OptorSim [24], our simulator generates multicriteria strategies as a number of supporting schedules for metascheduler reactions to the events connected with resource assignment and advance reservations. Strategies for more than 12000 jobs with *a fixed completion time* were studied. Every *task of a job* had *randomized* completion time estimations, computation volumes, data transfer times and volumes with a uniform distribution. These parameters for various tasks had difference which was equal to 2...3. Processor nodes were selected in accordance to their relative performance. For the first group of "fast" nodes the relative performance was equal to 0.66…1, for the second and the third groups 0.33…066 and 0.33 ("slow" nodes) respectively. A number of nodes was conformed to a job structure, i.e. a task parallelism degree, and was varied from 20 to 30.

The strategies types are:

- `S1` – with fine-grain computations and active data replication policy;
- `S2` – with fine-grain computations and a remote data access;
- `S3` – with coarse-grain computations and static data storage;
- `MS1` – with fine-grain computations, active data replication policy, and the best- and worst execution time estimations (a modification of strategy `S1)`.

The strategy `MS1` is less complete than the strategy `S1` in the sense of coverage of events in distributed environment. However the important point is the generation of a strategy by efficient and economic computational procedures of the metascheduler (see Fig. 1). The type `S1` has more computational expenses than `MS1`.

We have conducted the statistical research of the critical works method for application-level scheduling with above-mentioned types of strategies `S1`, `S2`, `S3`. The main goal of the research was to estimate a forecast possibility for making application-level schedules without taking into account independent job flows. For 12000 randomly generated jobs there were 38% admissible solutions for `S1` strategy, 37% for `S2`, and 33% for `S3` (Fig. 3, a). This result is obvious: application-level schedules implemented by the critical works method were constructed for available

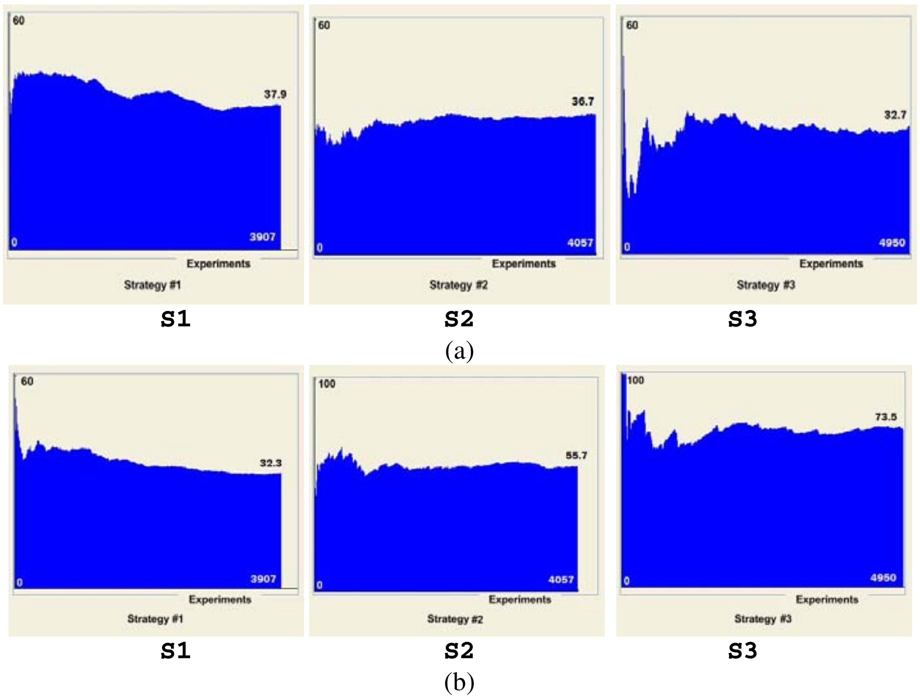


Fig. 3. Simulation results for application-level scheduling: percentage of experiments with admissible schedules (a) and percentage of collisions for "fast" processor nodes (b)

resources non-assigned to other independent jobs. Along with it there is a conflict distribution for the processor nodes that have different performance ("fast" are 2-3 times faster, than "slow" ones): 32% for "fast" ones, 68% for "slow" ones in S1, 56% and 44% in S2, 74% and 26% for S3 (Fig. 3, b). This may be explained as follows. The higher is the task state of distribution in the environment with active data transfer policy, the lower is the probability of collision between tasks on a specific resource.

In order to implement the effective scheduling and resource allocation policy in the virtual organization we should coordinate application and job-flow levels of the scheduling. For each simulation experiment such factors as job completion "cost" (Section 3), task execution time, scheduling forecast errors (start time estimation), strategy live-to-time (time interval of acceptable schedules in a dynamic environment) were studied (Fig. 4). Figure 4, a shows load level statistics of variable performance processor nodes which allows discovering the pattern of the specific resource usage when using strategies with coordinated job-flow and application-levels scheduling.

The strategy S2 performs the best in the term of load balancing for different groups of processor nodes, while the strategy S1 tries to occupy "slow" nodes, and the strategy S3 – the processors with the highest performance (see Fig. 4, a). Factor quality analysis of S2, S3 strategies for the whole range of execution time estimations for the selected processor nodes as well as modification MS1, when best- and

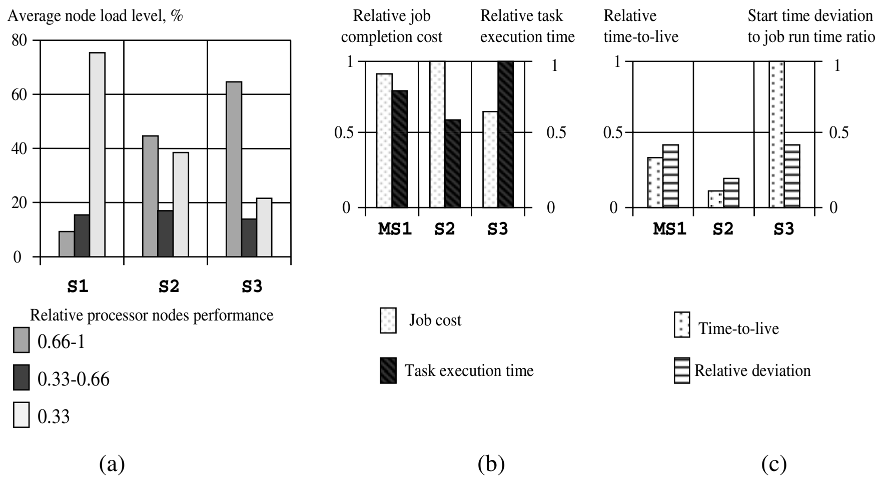


Fig. 4. QoS factors in diverse strategies: processor node load level (a); job completion cost and task execution time (b); time-to-live and start deviation time (c)

worst-case execution time estimations were taken, is shown in Figures 4, b and 4, c. Lowest-cost strategies are the “slowest” ones like `S3` (see Fig. 4, b), they are most persistent in the term of time-to-live as well (see Fig. 4, c). The strategies of the type `S3` try to monopolize processor resources with the highest performance and to minimize data exchanges. Withal, less persistent are the “fastest”, most expensive and most accurate strategies like `S2`. Less accurate strategies like `MS1` (see Fig. 4, c) provide longer task completion time, than more accurate ones like `S2` (Fig. 4, b), which include more possible events, associated with processor node load level dynamics.

5 Conclusions and Future Work

The existing works in scheduling problems are related to either job scheduling problems or application-level scheduling. Fundamental difference between them and the approach described is that the resultant dispatching strategies are based on the integration of job-flows management methods and application-level techniques. It allows increasing the quality of service for the jobs and distributed environment resource usage efficiency. Our results are promising, but we have bear in mind that they are based on simplified computation scenarios, e.g. in our experiments we use first-come-first-served (FCFS) management policy in local batch-job management systems. Afore-cited research results of strategy characteristics were obtained by simulation of global job-flow in a virtual organization. Inseparability condition for the resources requires additional advanced research and simulation approach of local job passing and local processor nodes load level forecasting methods development. Different job-queue management models and scheduling algorithms can be used (FCFS modifications, least-work-first (LWF), backfilling, gang scheduling etc.) here. Along with it local administering rules can be implemented. One of the most important aspects here is that advance reservations [17, 20] have impact on the quality of service. Some of the researches (particularly the one in Argonne National Laboratory) show, that preliminary reservation nearly always increases queue waiting time. Backfilling decreases this time. With the use of FCFS strategy waiting time is shorter than with the use of LWF. On the other hand, estimation error for starting time forecast is bigger with FCFS than with LWF. Backfilling that is implemented in Maui cluster scheduler includes advanced resource reservation mechanism and guarantees resource allocation. It leads to the difference increase between the desired reservation time and actual job starting time when the local request flow is growing. Some of the quality aspects and job-flow load balance problem are associated with dynamic priority changes, when virtual organization user changes execution cost for a specific resource. All of these problems require further research.

Acknowledgments. This work was supported by the Russian Foundation for Basic Research (grant no. 09-01-00095) and by the State Analytical Program "The higher school scientific potential development" (project no. 2.1.2/6718).

References

1. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Int. J. of High Performance Computing Applications 15(3), 200–222 (2001)
2. Thain, D., Tannenbaum, T., Livny, M.: Distributed Computing in Practice: the Condor Experience. Concurrency and Computation: Practice and Experience 17(2-4), 323–356 (2004)
3. Roy, A., Livny, M.: Condor and Preemptive Resume Scheduling. In: Nabrzyski, J., Schopf, J.M., Weglarz, J. (eds.) Grid resource management. State of the art and future trends, pp. 135–144. Kluwer Academic Publishers, Dordrecht (2003)
4. Krzhizhanovskaya, V.V., Korkhov, V.: Dynamic Load Balancing of Black-Box Applications with a Resource Selection Mechanism on Heterogeneous Resources of Grid. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 245–260. Springer, Heidelberg (2007)
5. Berman, F.: High-performance Schedulers. In: Foster, I., Kesselman, C. (eds.) The Grid: Blueprint for a New Computing Infrastructure, pp. 279–309. Morgan Kaufmann, San Francisco (1999)
6. Yang, Y., Raadt, K., Casanova, H.: Multiround Algorithms for Scheduling Divisible Loads. IEEE Transactions on Parallel and Distributed Systems 16(8), 1092–1102 (2005)
7. Natrajan, A., Humphrey, M.A., Grimshaw, A.S.: Grid Resource Management in Legion. In: Nabrzyski, J., Schopf, J.M., Weglarz, J. (eds.) Grid resource management. State of the art and future trends, pp. 145–160. Kluwer Academic Publishers, Dordrecht (2003)
8. Beiriger, J., Johnson, W., Bivens, H., Humphreys, S., Rhea, R.: Constructing the ASCI Grid. In: 9th IEEE Symposium on High Performance Distributed Computing, pp. 193–200. IEEE Press, New York (2000)
9. Frey, J., Foster, I., Livny, M., Tannenbaum, T., Tuecke, S.: Condor-G: a Computation Management Agent for Multi-institutional Grids. In: 10th International Symposium on High-Performance Distributed Computing, pp. 55–66. IEEE Press, New York (2001)
10. Abramson, D., Giddy, J., Kotler, L.: High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In: International Parallel and Distributed Processing Symposium, pp. 520–528. IEEE Press, New York (2000)

11. Ranganathan, K., Foster, I.: Decoupling Computation and Data Scheduling in Distributed Data-intensive Applications. In: 11th IEEE International Symposium on High Performance Distributed Computing, pp. 376–381. IEEE Press, New York (2002)
12. Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J.: Multicriteria Aspects of Grid Resource Management. In: Nabrzyski, J., Schopf, J.M., Weglarz, J. (eds.) Grid resource management. State of the art and future trends, pp. 271–293. Kluwer Academic Publishers, Dordrecht (2003)
13. Tracy, D., Howard, J.S., Noah, B., Ladislau, B., et al.: A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. J. of Parallel and Distributed Computing 61(6), 810–837 (2001)
14. Buyya, R., Abramson, D., Giddy, J., Stockinger, H.: Economic Models for Resource Management and Scheduling in Grid Computing. J. of Concurrency and Computation: Practice and Experience 14(5), 1507–1542 (2002)
15. Dail, H., Sievert, O., Berman, F., Casanova, H., et al.: Scheduling in the Grid Application Development Software project. In: Nabrzyski, J., Schopf, J.M., Weglarz, J. (eds.) Grid resource management. State of the art and future trends, pp. 73–98. Kluwer Academic Publishers, Dordrecht (2003)
16. Anderson, D.P., Fedak, G.: The Computational and Storage Potential of Volunteer Computing. In: IEEE/ACM International Symposium on Cluster Computing and Grid, pp. 73–80. IEEE Press, New York (2006)
17. Ioannidou, M.A., Karatza, H.D.: Multi-site Scheduling with Multiple Job Reservations and Forecasting Methods. In: Guo, M., Yang, L.T., Di Martino, B., Zima, H.P., Dongarra, J., Tang, F. (eds.) ISPA 2006. LNCS, vol. 4330, pp. 894–903. Springer, Heidelberg (2006)
18. Tang, M., Lee, B.S., Tang, X., Yeo, C.K.: The Impact of Data Replication on Job Scheduling Performance in the Data Grid. Future Generation Computing Systems 22(3), 254–268 (2006)
19. Dang, N.N., Lim, S.B., Yeo, C.K.: Combination of Replication and Scheduling in Data Grids. Int. J. of Computer Science and Network Security 7(3), 304–308 (2007)
20. Aida, K., Casanova, H.: Scheduling Mixed-parallel Applications with Advance Reservations. In: 17th IEEE International Symposium on High-Performance Distributed Computing, pp. 65–74. IEEE Press, New York (2008)
21. Toporkov, V.: Multicriteria Scheduling Strategies in Scalable Computing Systems. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 313–317. Springer, Heidelberg (2007)
22. Toporkov, V.V., Tselishchev, A.S.: Safety Strategies of Scheduling and Resource Co-allocation in Distributed Computing. In: 3rd International Conference on Dependability of Computer Systems, pp. 152–159. IEEE CS Press, Los Alamitos (2008)
23. Toporkov, V.V.: Supporting Schedules of Resource Co-Allocation for Distributed Computing in Scalable Systems. Programming and Computer Software 34(3), 160–172 (2008)
24. William, H.B., Cameron, D.G., Capozza, L., et al.: OptorSim – A Grid Simulator for Studying Dynamic Data Replication Strategies. Int. J. of High Performance Computing Applications 17(4), 403–416 (2003)

Filmification of Methods: Representation of Particle-In-Cell Algorithms

Yutaka Watanobe[1], Victor Malyshkin[2], Rentaro Yoshioka[1], Nikolay Mirenkov[1], and Hamido Fujita[3]

[1] University of Aizu, Japan
[2] Institute of Computational Mathematics and Mathematical Geophysics, Russia
[3] Iwate Prefectural University, Japan

Abstract. Filmification of methods is an approach to find new formats for program and data/knowledge representation. It is also to create a basis for specifying and developing a new generation of programming environments. Within this approach various algorithms are analyzed and represented as cyberFilms where special visual super-symbols (icons) are introduced for defining meaning of the cyberFilm frames. In this paper, the filmification of methods is applied for particle-in-cells algorithms. Results demonstrate a promising compactness of the program representation, covering technical details of parallel implementation and an effectiveness of an open set of icons.

Keywords: cyberFilms, particle-in-cell, visual languages, programming.

1 Introduction

In spite of great efforts and essential progress of software industry, still we can say that our programming languages are good for computers, but they are not so good for people communication and their understanding of programs. It is possible to regularly read reports that American companies lose about $60 billion every year because of software deficiency. On the other hand, our future is getting to be much more difficult because PCs are becoming multi- and many-core super-computers and super-computers are becoming peta-flops machines. In addition, a variety of embedded systems is growing, global business and knowledge integration is enhancing, software security & safety are worsening, etc. Recently, a serious number of projects has been initiated by Big computer players to develop new programming tools oriented to multi- and many-core architectures (at University of California Berkeley [16], University of Illinois [18], Stanford University [17] and Technical University of Munich [19] to mention a few). They are in addition to existing Fujitsu remote procedure calls, RapidMind's Multicore development platform, PeakStream's math library for graphics processors, Nvidia's CUDA, etc. Great efforts are taken to acquire knowledge of efficient implementations of software components and to introduce abstraction mechanisms that hide hardware details from developers. Within these efforts, a variety of approaches can be used. However, we think that the acquisition should not pay attention only to technical aspects of implementations. High level representations of algorithmic features easier understandable by application programmers should also be collected. In other words, we should promote not only high-performance of parallel computers, but also high-performance of people involved.

In this paper we are promoting a concept of algorithmic cyberFilms as a new environment for high-level specification (programming) of algorithms, acquisition them in a format of four visual languages supported by template programs responsible for efficient implementations of corresponding high-level constructs. One of the goals of the approach is to allow application programmers to focus on their models and simulation experiments rather than on problems of efficient implementation. Within this concept an open set of super-symbols (icons) to define contents of cyberFilm frames is applied. However, new icons, to be introduced, should represent some knowledge of a class of algorithms and methods of their implementation. To test the idea of the open set and applicability of the concept for various computational methods, features of particle-in-cells algorithms and an attempt to create a cyberFilm for them are presented.

The use of cyberFilms to represent algorithms and to develop programs can be considered as a type of best practice for developing parallel, as well as sequential, programs. Users are guided to explicitly specify a pre-defined set of features of their programs and related algorithms in such a way that they can later be individually accessed by compilers to exploit parallelism and other optimizations. Furthermore, the features can be browsed individually, or collectively, by specially tailored visual languages for enhancing users' comprehension to support bringing out extra creativity.

The rest of the paper is organized as follows. Algorithmic cyberFilm concept and related papers are considered in Section 2, some features of particle-in-cells (PIC) algorithms are presented in Section 3, filmification of the PIC approach based on, so called, algorithmic skeleton view and integrated view are provided in Section 4, and finally some additional comments and conclusion are presented in Sections 5 and 6.

2 Algorithmic CyberFilm Concept and Related Papers

A cyberFilm is a set of series of color pictures (frames). A picture is to represent a view (a feature) of an object or process. A series of pictures is to represent a multiple view (a number of features) of an object or process. A multiple view is to make the corresponding object or process be self-explained. The self-explained film means that the associated pictures are organized and presented in such a way that the semantic richness of data/ knowledge is clearly brought out. A number of features presented by multimedia frames are assembled into a structure (cyberFilm). In the right time a right group of corresponding frames is extracted and presented through a right channel. Fig. 1 depicts a structure scheme of the cyberFilm format.

The first (leftist) series of frames represents algorithmic skeletons that show space data structures and temporal schemes of computational flows on these

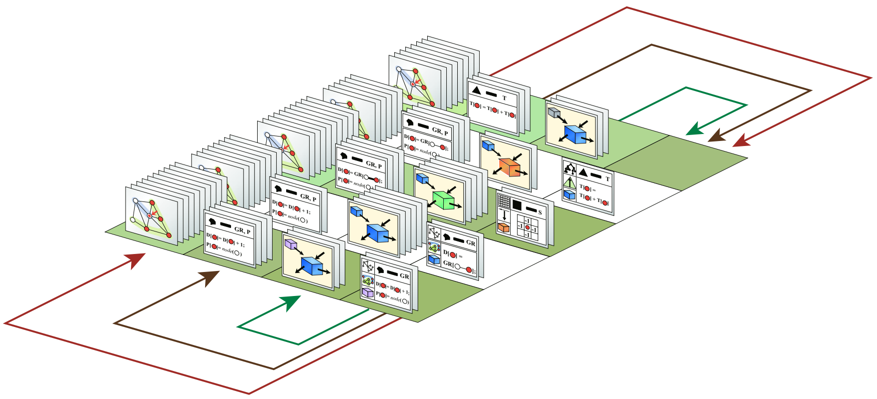


Fig. 1. Four series of frames representing four views of an algorithm

structures. The second series shows variables and formulas (actions) that are attached to the spice-time points of the algorithmic skeletons. The third series represents input/output operations that define the algorithmic interface with external world, as well as how a software component based on the algorithm should look from outside. Finally, the fourth series shows a compact combination of main features presented in the above mentioned groups of frames. For each series of frames there is its own visual language. As a result, the programming environment for making cyberFilms is a cluster of four mutually supplemented languages (and supporting systems) to define four different views of each software component. Each language uses its own set of pictures and super-characters of a self-explanatory type. The following abbreviations are used for these visual languages:

Algorithmic cyberFilm language = {LAD, LAC, LAF, LAT}, where

– LAD - Language of Algorithmic Dynamics
– LAC - Language of Algorithmic Commands
– LAF - Language of Algorithmic interFace
– LAT - Language of Algorithmic ”Text” (language of integrated views).

In these languages, open sets of visual constructs and super-characters are used to define the contents of the cyberFilm frames and their translation into executable codes. In addition to the constructs and super-characters, special background pictures and symbols are used to simplify the contents understanding. They do not have any influence on the code generation, but on user’s recognition of semantics.

Examples of such constructs, super-characters and background pictures, and the descriptions of various aspects of the filmification of methods related to sequential and parallel matrix multiplications, solving algebraic and partial differential equations, cellular automation-like algorithms, as well as to algorithms on trees, pyramids, etc. can be found in [1-6]. A large set of algorithms on graphs and a corresponding library are presented in [7-8]. Some aspects of the visualization of input/output operations are considered in [9]. Fig.2 depicts examples of icons to represent traversal schemes of algorithms. In fact, they are super-characters for cyberFilm frames of LAT. Some of them are to specify parallel schemes of computation; others are to specify sequential ones. For example, meaning of the top-left icon is shown at Fig.3 where parallel operations are defined on nodes of columns traversed from left to right and back.

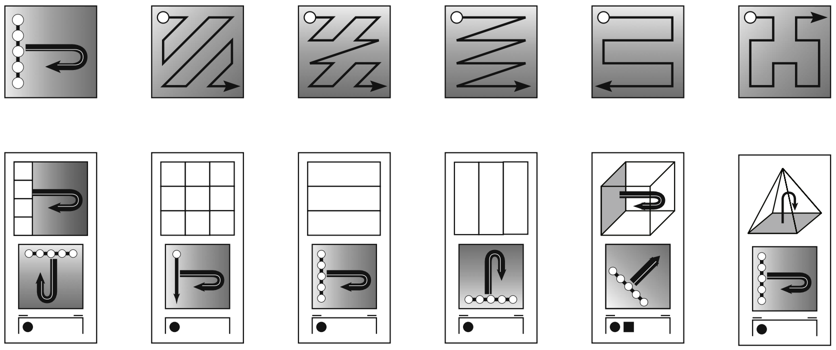


Fig. 2. Examples of icons to represent traversal schemes of algorithms

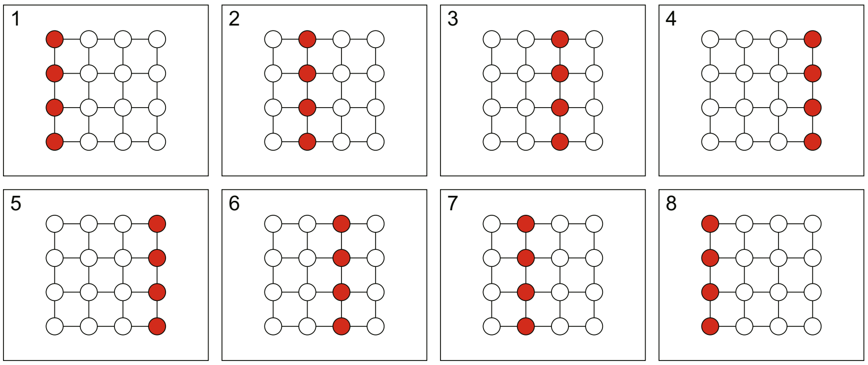


Fig. 3. LAD cyberFrames representing meaning of the top-left icon in Fig.2

For better understanding of the approach, let us make an illustrative look at a graph algorithm for solving the shortest path problem. It has only one LAT frame presented by Fig.4 (explanation details can be find in [8]). The first row of the frame is for data structures and variable declarations. It says that a graph structures is considered with 1000 nodes and edges available from a file of A. Then variable D of the integer type with elements attached to each node of the graph is declared (the first micro-icon is for the graph structure, and the second is for integer). After that, variable GR of integer type with elements attached to each edge is declared. Under the first row, the algorithm itself is presented. The left icon represents it by requiring the definition of two operations. One is a terminal (represented by a circle and an executable formula) and another is non-terminal (represented by square and another icon). In this case, the additional icon requires the definitions of three terminal operations. An idea behind the terminal operations is “do where highlighted nodes are.”

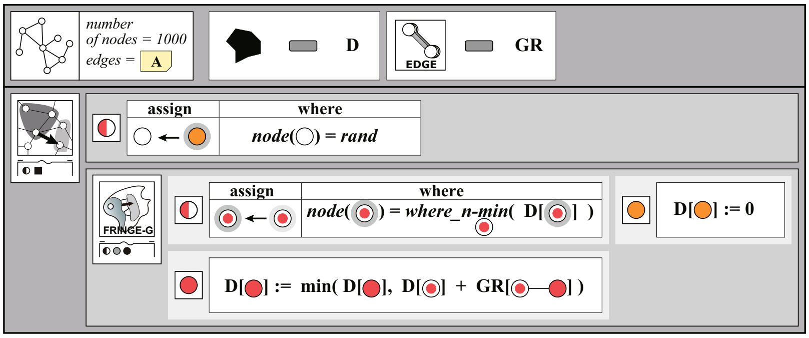


Fig. 4. LAT cyberFilm frame for the shortest path problem

If user does not know/remember meaning of icons in such a view, the system can immediately explain them by a set of LAD cyberFilm frames (like in Fig. 5) where differently highlighted nodes mean different types of operations.

Basic types of the operation highlighting used in LAD cyberFilm frames are depicted by Fig. 6.

For our further consideration, the following flashing operations (a, b, c and f) are important to mention:

– **a** (full highlighting) is to specify possible change of variables in a node of the data space structure,
– **b** (contour highlighting) is to specify reading access to variables in a node,
– **c** (half highlighting) is to specify (alternative) activities on nodes for the next step of algorithm implementation,
– **f** (doubled highlighting) is to specify a transformation of the data space structure.

These highlightings are to focus on types of operations (activities); the operations themselves are defined by LAC and LAT frames.

In order to transfer our results into a basis for a breakthrough technology, within each set of cyberFrames above mentioned, we pay special attention to the following:

– Self-explanatory features of each visual symbol, frame, scene and cyberFilm as a whole to decrease necessity of rote memorization, unnecessary mental simulation, and people adaptation to technology.

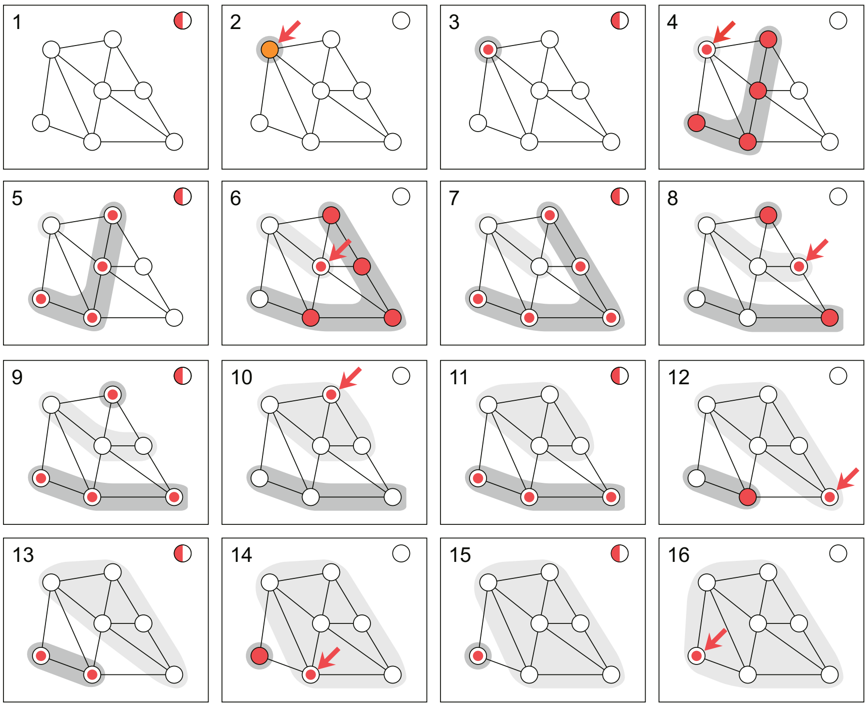


Fig. 5. LAD cyberFilm frames representing meaning of icons from Fig. 4

	symbol	animation	name	operation	write	read	control
(a)	●	● ⇌ ○	full	change (write)	○	○	
(b)	◉	◉ ⇌ ◉	contour	take (read)		○	
(c)	◐	◐ ⇌ ○	half	select	○	○	○
(d)	⊙	⊙ ⇌ ○	center	wait		○	○
(e)	●	● ⇌ ○	wave	continue	○	○	○
(f)	●●	●● ⇌ ○	full-double	transform	○	○	○
(g)	◐◐	◐◐ ⇌ ○	half-double	transform-select	○	○	○
(h)	●	● ⇌ ○	bar	select with break	○	○	○

Fig. 6. Basic types of the operation highlighting (flashing) used in cyberFilm frames

– Acquisition of data-knowledge as cyberFilms to increase the programmer performance, and as template programs to guarantee high performance of executable code on parallel architectures.
– Programming each software component in four picture-based languages to reach high level reliability through automatic checking correctness and human recognition of the component meaning.

The use of background images for cyberFrames to support not only understandability language constructs, but also to express such things as beauty or feeling. A fundamental feature of the approach (as we have mentioned in the introduction) is a concept of an open set of super-symbols (icons) to define contents of cyberFilm frames and manage the visual code compactness. However, to be introduced, new icons should represent some special knowledge of a class of algorithms and methods of their implementation. That is why this paper is related to features of particle-in-cells algorithms and our attempt to create a cyberFilm for them.

3 PIC Algorithm Features

Particle-In-Cell (PIC) is widely used method for application models where essential irregularity and even dynamically changed irregularity of the data structures should be involved. PIC is described in many papers and books, see for example [10, 11], where numerous references to the PIC implementations can be found. Let us look at some general features of the method through considering the problem of energy exchange in plasma cloud. A real physical space is represented by a model of a simulation domain called the space of modeling (SM). SM contains the test particles; each particle is described by the 3D coordinates, velocity, charge and mass. The electric E and magnetic B fields are defined as vectors and discretized upon rectangular mesh (Fig.7 and Fig.8). At any moment of modeling a particle belongs to a certain cell SM. The trajectories of a huge number of test particles are calculated as these particles are moved under the influence of the electromagnetic fields computed self-consistently on a discrete mesh. These trajectories represent a desirable solution of the system of differential equations describing a physical phenomenon under study.

The dynamics of the plasma cloud is determined by integrating the equations of motion of every particle in the series of discrete time steps. At each time step $t_{k+1} := t_k + \Delta t$ the following is done:

1. For each particle, the Lorentz force is calculated from the values of electromagnetic fields at the nearest mesh points (gathering phase);
2. For each particle the new co-ordinates and velocity of a particle are calculated; a particle can move from one cell to another (moving phase);

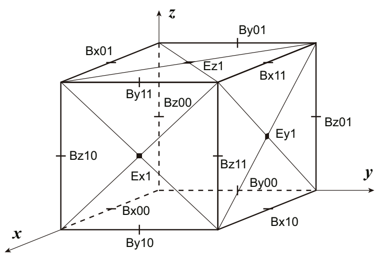


Fig. 7. A cell of the SM with the electric E and magnetic B fields, discretized upon a mesh

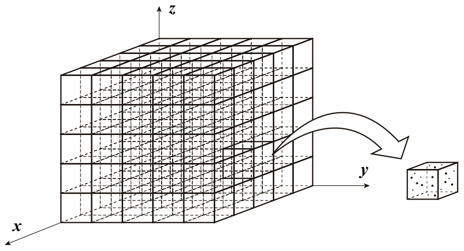


Fig. 8. The whole space of modeling (SM) assembled out of cells

3. For each particle the charge carried by a particle to the new coordinates is calculated to obtain the current charge and density, which are also discretized upon the rectangular mesh (scattering phase);
4. Maxwell's or Poisson equations are solved to update the electromagnetic field (mesh phase).

Fragmentation and dynamic load balancing are the key features of the PIC parallel implementation. For the PIC parallelization in each PE a number of rectangular blocks (sub-domains of SM), including electromagnetic fields at the corresponding mesh points and particles in the corresponding cells can be loaded for processing (technologically the use of equal size blocks can be better). If a particle leaves its sub-domain on the second step and flies to another sub-domain in the course of modeling, then this particle should be transferred to the PE containing this latter sub-domain (particles migration). Thus, even with an equal initial workload of the PEs, in several steps of modeling, some PEs might contain far more particles than the others. This results in the load imbalance. If the load imbalance exceeds a threshold, then some block(s) should leave overloaded PE and migrate to a neighbor under-loaded PE (dynamic load balancing). Many algorithms of dynamic load balancing were published [12,13]. So, our goal is to take into account some of their features for possible representations by constructs of the cyberFilm language.

3.1 Description of the Forces Distribution Scheme on the Planes and Inside a Cell

There are different schemes of the field discretization. One of them (electric and magnetic fields discretization) is shown in Fig.7. Another scheme of the gravitational field discretization can be found below in Fig. 9 (from [14]). This scheme is implicitly used in algorithms for forces values calculation inside a cell (interpolation) that depends on a model. These algorithms and forces distribution should be programmed.

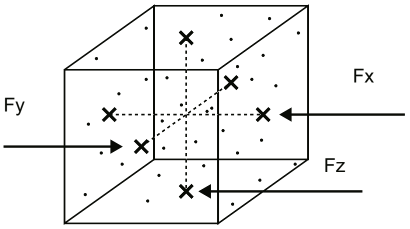


Fig. 9. Gravitational field discretization

3.2 Virtual Cells

Virtual cells are implemented in substantially different way then ordinary cells. The method and algorithms of virtual cells implementation should be included into compiler.

3.3 Particles and Processes Migration

Different algorithms can be used for processes migration implementation in the case some node is overloaded because the most simple and natural algorithm of cells migration consumes too many resources. Several algorithms of cells migration are used when the cells are aggregated into layers and columns [15]. Also migration should be planned in such a way in order to predict the development of simulated processes. The notion of threshold and algorithm of its calculation should be provided.

3.4 Data Input/Output and Algorithms of Initial Data Distribution

Initial cells and particles distributions among the multicomputer nodes should be also programmed because this distribution depends on the initial state of a model.

3.5 Mutual Exclusion

In the model there are usually operations that demand the use of global information. In particular, the invariable value of energy means that the model is valid yet. The energy value is calculated as the sum of energy values in each cell. Therefore, the synchronization programming for operations mutual exclusion should be provided.

3.6 Gathering-Sending of Particles

Some particles should migrate from one to neighbor cells in the course of simulation. This is done after synchronization. Therefore, the particles transfer should be programmed.

3.7 Dividing-Incorporation of the Virtual Cells

A cell, in which the number of particles is too big, i.e., all its particles cannot be located inside the memory of one node of multicomputer, is divided into several cells that are implemented as virtual cell. Any virtual cell in its turn also can be divided into several cells. If particles leave a virtual cell then some adjacent virtual cells can be incorporated. The algorithms of cells dividing and incorporation should be programmed in the cyberFilm language.

4 Filmification of Particle-In-Cells

Four series cyberFilm frames and corresponding four visual languages should be used to represent an algorithm. However, black-and-white printing materials are not the best way to show the filmification approach. So, we will focus on the integrated view (in the LAT language) and on the algorithmic skeleton view (in the LAD language) clarifying meaning of icons in the cyberFilm frames from the integrated view. Fig.10 is depicted a LAT cyberFilm frame for a Particles-In-Cells algorithm (color versions of this and other figures related to PIC algorithmis are presented at: http://borealis.u-aizu.ac.jp/aks/film/particle.html). To be inside one page figure (and in one cyberFilm frame of the integrated view), we omitted the declaration of variables (the top row of Fig.4 shows an example of such type declaration) and made some generalization of formulas involved. The variables omitted include 1) arrays related to nodes of 3D mesh structure, to cell centers, and to cell side centers, as well as 2) sets of particles and their attributes. It is also assumed that within this declaration, values of gravity forces in the cell side centers were defined.

LAT frames usually include two columns: the first column is to represent an algorithm hierarchy through a set of high-level visual constructs (icons) and non-terminal operations, and the second one is to represent terminal operations (formulas) disclosing the non-terminal operations. In our case, the hierarchy of the first column includes only two levels. The top icon represents the PIC algorithm as a whole. It shows that seven non-terminal operations should be defined (squares/cubes and circles are used for non-terminals and terminals, respectively). The bottom part of the icon also shows that the operations are hierarchically involved in two internal constructs allowing some parallel activity. Very short lines above these internal constructs (at the left and right sides) are hints about the parallel activity. Algorithmic skeleton view for this icon is presented by Fig.11, where seven cyberFilm frames (computational steps) represent the PIC algorithm. Frame 6 is responsible for branching the computation. All other frames highlight parallel non-terminal activity on all cells of 3D mesh structure. The activities are different in different frame and highlighted by different colors (here, different colors are pointed by different numbers near 3D mesh structures). The parallel activity of each frame should be barrier synchronized before going to the next frame (a special link is depicted if we want to avoid the barrier synchronization between frames). Within each frame, the parallel activity on cells is defined as identical (the same color is used for all cells). To show

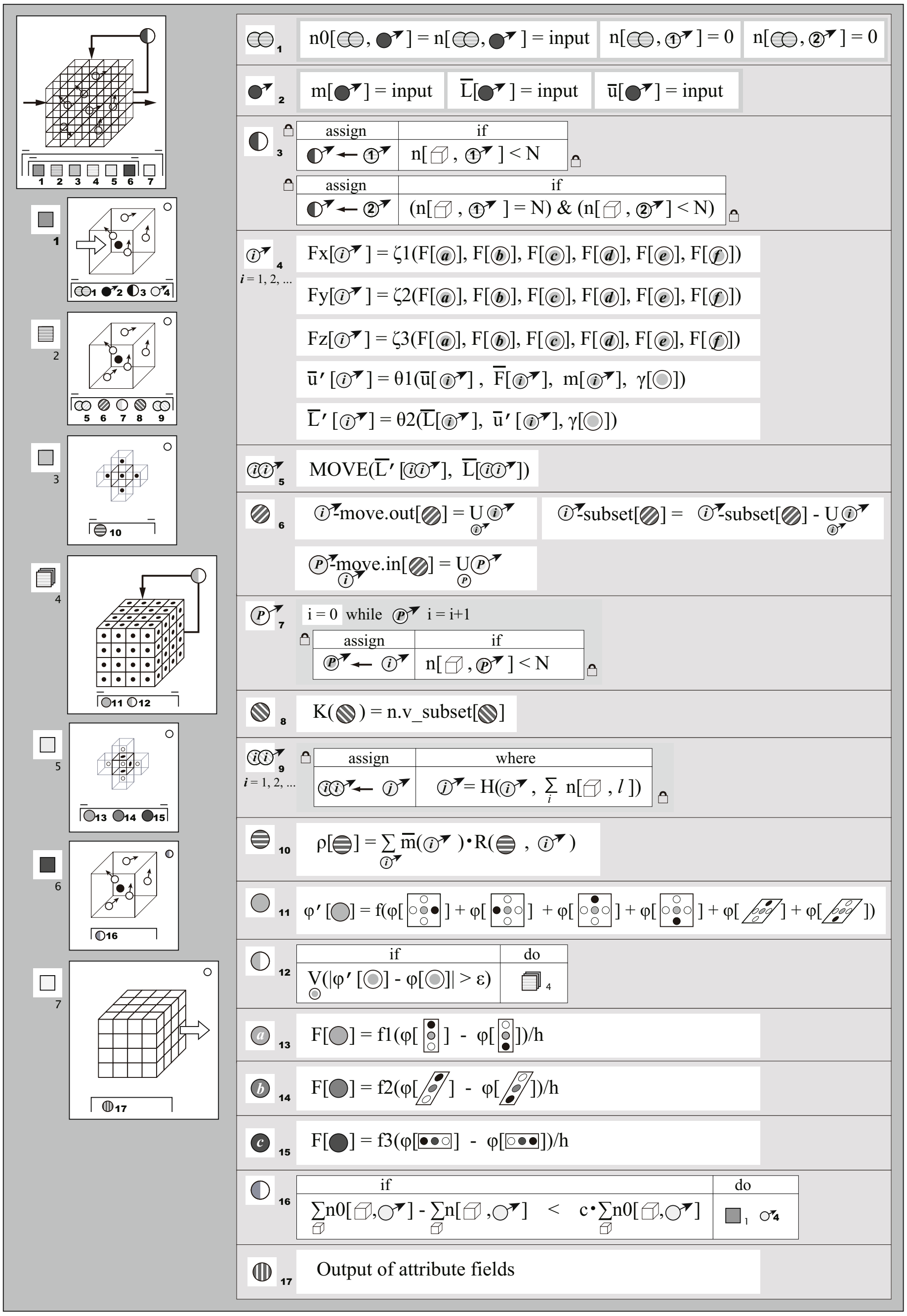


Fig. 10. LAT cyberFilm frame for a Particles-in-cells algorithm

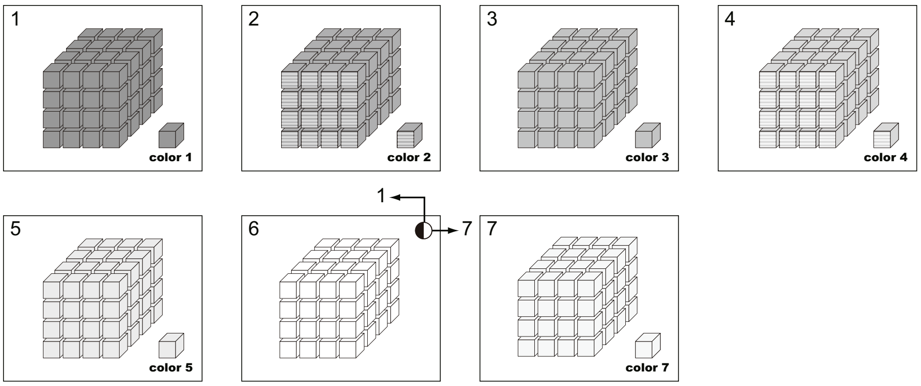


Fig. 11. LAD cyberFrames of PIC algorithm and the top-left icon from Fig. 10

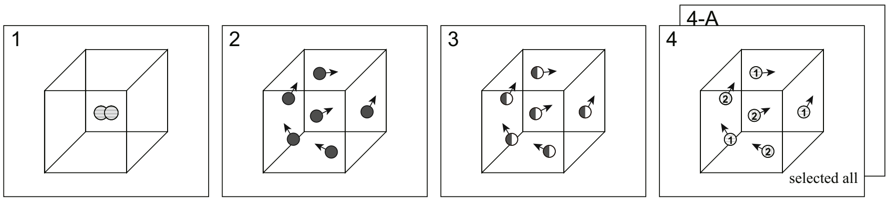


Fig. 12. LAD cyberFrames of non-terminal activity 1 and corresponding icon from Fig. 10

different activity, different colors are usually applied. In our case, the activity on boundary cells and internal cells can be shown by different colors. For simplify, this time we consider that this differentiation can be done on the terminal formula level.

On the second level of the PIC hierarchy, seven non-terminal activities above mentioned are presented by a set of other icons depicted in the first column. The new icons are located in the first column with a same shift to the right. These icons represent some visual constructs requiring the definition of only terminal operations (if non-terminals are needed, they should be explained by beneath icons shifted to the left). Non-terminal activity 1 requires the definition of four terminal operations. The algorithmic skeleton view of the non-terminal activity is presented by Fig.12.

It says that the activity includes four steps of the following type terminals: 1) - operations on variables attached to the cell center to create/change a dynamical data structure (a set of particles), 2) - parallel operations on variables attached to different particles, 3) - parallel decisions on each particle status, and 4) - parallel operations on variables attached to particles with possible attention to the existence of particle subsets. Some hints about results of operations at the first frame can be observed at the second frame, and results of operations at the third frame can be observed at the fourth frame. These hints are included into semantic of doubled and half-flashing operations. The precise formulas for all frame operations are presented in the rows of the second column of the LAT view. Let us look at these formulas. They have the standard format of left and right parts, but not so standard index expressions which rather directly show where (in the space structures) data should be taken from, and where variables should be changed. This show is based on flashed (highlighted) nodes and index stencils. For PIC algorithms to show operations on particles, special visual symbols (circles with arrow) have been introduced.

Formulas for frame 1 are related to variables **n** and **n0** attached to cell centers and declared to represent sets of particles involved (**n0** is to save the initial number of the particles); different subsets are specified by different types of particles and different cells where particles are considered. First, variables **n** and **n0** obtain their value through an input operation (in fact, a subset of particles is created), then two empty subsets of type 1 and type 2 particles are created. The doubled flashing node is to specify a transformation of the data space structure (in this case, an appearance of particles); a hint about a type of the transformation is presented by the next frame.

Formulas for frame 2 are related to variables attached to particles: **m** (mass), and vectors **L** (location) and **u** (velocity).

Formulas for frame 3 are about decisions related to particle status (activity at the next frame). In this case, a distribution of the initial set particles among two subsets if the particle number in a cell is greater than a threshold of N; a cube micro image is an index expression to point a cell under consideration, particle symbols with I and 2 are to refer to particles of type 1 and 2, respectively. Clock micro icons are to define indivisible actions for checking conditions and the allocation of a particle to a subset. A hint about possible results of the decisions is presented by the next frame (”selected all” is to point that all particles change the status).

Formulas for frame 4 are to calculate (for both types of particles) external forces on each particle, new velocity and location. Contour flashing nodes of **a**, **b**, **c**, etc. types, as well as contour flashing particle of **i** type are to show that data are taken from the cell side centers and from particles themselves. However, the contour flashings are presented only in the formulas and are not in frame 4 that displays only full flashing particles where variables should be updated. To provide some additional explanations of space structure places for reading data, an auxiliary frame (Fig.13) behind frame 4 is used and shown on request. (For simplicity, here schemes of operations are presented instead of real operations.) Now let us go to the next non-terminal activity of the algorithm representation. Non-terminal activity 2 requires the definition of five terminal operations. Corresponding frames of the algorithmic skeleton view are presented by Fig.14.

The first frame says that a parallel transformation operation should be done on each particle. *The (terminal) operation* for this frame is defined by procedure

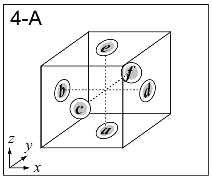


Fig. 13. An auxiliary frame 4-A for Fig. 12

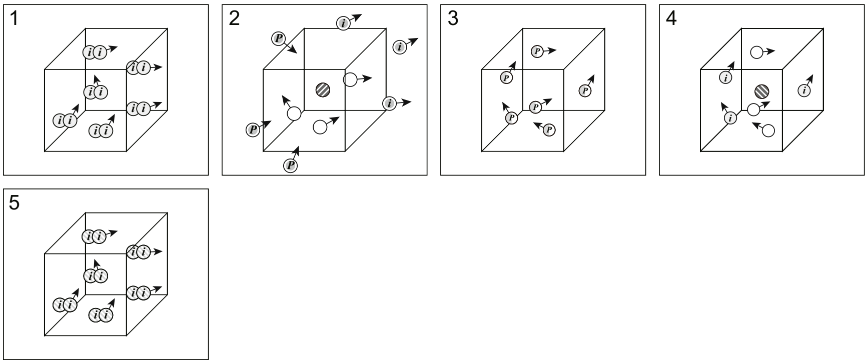


Fig. 14. Five algorithmic skeleton frames for non-terminal activity 2

MOVE on two arguments (current and expected positions of a particle). Results of the transformation can be seen at frame 2. Some particles of type i ($i = 1, 2, ...$) have been moved out of cell, but some of type p are coming into the cell from other cells. *The formulas* related to frame 2 calculate subsets of particles moved out, subsets of particles still stayed in, and a subset of particles came from other cells. *The operation* for frame 3 makes decisions on distribution of new comers among subsets of type i particles. First, p-particles go to a subset with $i = 1$, then, if it is full and p-particles exist, to a subset with $i = 2$, and so on. The operation for frame 4 calculates K (the number of subsets related to particles with different i, in fact, it is the number of virtual cells mentioned in Section 3.7). We do not use explicitly this value in further formulas, because symbol $i = 1, 2, ...$ does it implicitly. However, we think it is useful to show as possible opportunity. Finally, the operation for frame 5 transforms the number of the subsets into smaller one if they are rather empty. The calculation of the new subset number and assigning particles in the subsets are performed on each particle in parallel based on function H. Checking conditions and assigning a particle are performed as a non-divisible operation.

In a similar way, we can consider non-terminal activity of further five steps. For example, activity 3 is to calculate some physical field values in centers of neighboring cells through contribution from each particle belonging to a cell under consideration (see, terminal formula 10). Activity 4 is to solve a Poisson equation based on terminal operations 11 and 12. Activities 5 and 6 are to calculate some physical field values in three cell side centers and to check the termination of computation based on the number of particles left the physical space simulated (see, terminal formulas 13-16). Finally, Activity 7 is to specify output of attribute fields.

5 Additional Comments

In fact, it is our first attempt to apply the concept of cyberFilms for PIC algorithms. The frames of two views and super-symbols introduced for the frames show that the approach can be really promising for this types of algorithms. It is possible to say that the size (compactness) of the LAT programs can be very close to the compactness of formulas used by algorithms. The visual constructs presented can allow application programmers to focus on their models and methods rather than on technical details of parallel implementations. However, it does not mean that nobody should care about such implementations. In this approach, usually for each super-symbol representing a non-terminal activity, a set of template programs is created and saved in special library. This set is based on acquiring knowledge of efficient implementations depending on features of possible parallel architectures, sizes of the physical space and the particle number, as well as granularity of terminal operations involved. One temple program is selected from the set during the code generation.

The PIC algorithms possess a good natural parallelism related to operations in different cells and on different particles. However, the dynamics of load balancing requires a special attention to temple program implementations. In general, it can be embedded into such program implementations through, for example, the use of the virtual cell technique [17]. As a result, the user should not think about this problem at all. However, it is possible to assume cases where some attention to implementations within the high-level representation can be very useful for generating a more efficient executable code. That is why to show how it can be done, in the LAT program from Section 4 we introduced subsets of particles and presented operations on the subset dividing and merging. In fact, it is a rather direct form of the virtual cell representation.

6 Conclusion

An approach based on algorithmic cyberFilms has been applied to a high-level programming (specification) of particle-in-cell (PIC) algorithms. Language of algorithmic "text" (LAT) based on a set of super-symbols has been used to present an algorithm in a very compact form. Language of algorithmic dynamics (LAD) has been used to present explanation of the super-symbols mentioned. To make this high-level programming, a number of new super-symbols (icons) has been introduced. This introduction has also demonstrated that the concept of an open set of icons is a workable approach in acquiring new knowledge about algorithms and developing a new generation of programming environments. High-level constructs and the program as a whole have shown that application programmers can focus on their models and simulation experiments rather than on problems of the efficient implementation. In the example considered, the users can change the size of the 3D mesh structure, types of possible particles, attributes related to particles, physical fields, input of particles, visualization of results, etc. Though special template programs behind the constructs are considered as a main way of implementation knowledge acquisition, nevertheless, the constructs themselves can be used, if necessary, for providing serious support to the efficient code generation.

Our future work will be related to further analysis of PIC algorithms and corresponding template programs, as well as to analysis of other classes of application methods. The goal of the analysis is to prepare a fundamental basis for the specification of programming environment of a new generation.

References

1. Yoshioka, R., Mirenkov, N.: Visual Computing within Environment of Self-explanatory Components. Soft Computing Journal 7(1), 20–32 (2002)
2. Yoshioka, R., Mirenkov, N.: A Multimedia System to Render and Edit Self-explanatory Components. Journal of Internet Technology 3(1), 1–10 (2002)
3. Ebihara, T., Mirenkov, N., Nomoto, R., Nemoto, M.: Filmification of methods and an example of its applications. International Journal of Software Engineering and Knowledge Engineering 15(1), 87–115 (2005)
4. Saber, M., Mirenkov, N.: A visual representation of cellular automata-like systems. Journal of Visual Languages and Computing 15, 409–438 (2004)
5. Ebihara, T., Mirenkov, N.: Self-explanatory software components for computation on pyramids. Journal of Three Dimensional Images 14(4), 158–163 (2000)
6. Hirotomi, T., Mirenkov, N.: Multimedia representation of computation on trees. Journal of Three Dimensional Images 13(3), 146–151 (1999)
7. Watanobe, Y., Mirenkov, N., Yoshioka, R.: Algorithm Library based on Algorithmic CyberFilms. Journal of Knowledge-Based Systems 22, 195–208 (2009)
8. Watanobe, Y., Mirenkov, N., Yoshioka, R., Monakhov, O.: Filmification of methods: A visual language for graph algorithms. Journal of Visual Languages and Computing 19(1), 123–150 (2008)
9. Roxas, R., Mirenkov, N.: Input/Output Specifications within Self-explanatory Components. Journal of Three Dimensional Images 16(1), 129–134 (2002)
10. Grigoryev, Yu.N., Vshivkov, V.A., Fedoruk, M.P.: Numerical "Particle-in-Cell" Methods: Theory and applications. Utrecht-Boston (2002)
11. Hockney, R., Eastwood, J.: Computer Simulation Using Particles. McGraw-Hill Inc., New York (1981)
12. Kraeva, K., Malyshkin, V.: Dynamic load balancing algorithms for implementation of PIC method on MIMD multicomputers. Programmirovanie 1, 47–53 (1999) (in Russian)
13. Corradi, A., Leonardi, L., Zambonelli, F.: Performance Comparison of Load Balancing Policies based on a Diffusion Scheme. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300. Springer, Heidelberg (1997)

14. Kraeva, M.A., Malyshkin, V.E.: Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. Int. Journal on Future Generation Computer Systems 17(6), 755–765 (2001)
15. Kireev, S.: Parallel realization of PIC application to modeling of the problems of gravitational space dynamics. Autometriya 3, 32–39 (2006) (in Russian Parallel'naya metoda chastits v yacheikakh dlya modelirovaniya padach gravitatsionnoi kosmodynamiki)
16. Parallel Computing Laboratory, Univ. of California Berkeley, http://parlab.eecs.berkeley.edu/
17. Pervasive Parallelism Lab., Stanford University, http://ppl.stanford.edu/
18. Universal Parallel Computing Research Center, Univ. of Illinois, http://www.upcrc.illinois.edu/
19. The Munich Multicore, http://www.lrr.in.tum.de/~weidendo/mmi/doku.php

Parallel Evidence Propagation on Multicore Processors[*]

Yinglong Xia[1], Xiaojun Feng[3], and Viktor K. Prasanna[1,2]

[1] Computer Science Department
[2] Department of Electrical Engineering
University of Southern California, Los Angeles, CA 90089, U.S.A.
[3] Department of Computer Science and Technology
Tsinghua University, Beijing 100084, China
{yinglonx,prasanna}@usc.edu, fxj05@mails.tsinghua.edu.cn

Abstract. In this paper, we design and implement an efficient technique for parallel evidence propagation on state-of-the-art multicore processor systems. Evidence propagation is a major step in exact inference, a key problem in exploring probabilistic graphical models. We propose a rerooting algorithm to minimize the critical path in evidence propagation. The rerooted junction tree is used to construct a directed acyclic graph (DAG) where each node represents a computation task for evidence propagation. We develop a collaborative scheduler to dynamically allocate the tasks to the cores of the processors. In addition, we integrate a task partitioning module in the scheduler to partition large tasks so as to achieve load balance across the cores. We implemented the proposed method using Pthreads on both AMD and Intel quadcore processors. For a representative set of junction trees, our method achieved almost linear speedup. The execution time of our method was around twice as fast as the OpenMP based implementation on both the platforms.

Keywords: Exact inference, Multicore, Junction tree, Scheduling.

1 Introduction

A full joint probability distribution for any real-world system can be used for inference. However, such a distribution increases intractably with the number of variables used to model the system. It is known that independence and conditional independence relationships can greatly reduce the size of the joint probability distributions. This property is utilized by *Bayesian networks* [1]. Bayesian networks have been used in artificial intelligence since the 1960s. They have found applications in a number of domains, including medical diagnosis, consumer help desks, pattern recognition, credit assessment, data mining and genetics [2][3][4].

Inference in a Bayesian network is the computation of the conditional probability of the *query* variables, given a set of *evidence* variables as the knowledge

[*] This research was partially supported by the U.S. National Science Foundation under grant number CNS-0613376. NSF equipment grant CNS-0454407 is gratefully acknowledged.

to the network. Inference in a Bayesian network can be *exact* or *approximate*. Exact inference is NP hard [5]. The most popular exact inference algorithm for multiply connected networks was proposed by Lauritzen and Speigelhalter [1], which converts a Bayesian network into a *junction tree*, then performs exact inference in the junction tree. The complexity of exact inference algorithms increases dramatically with the density of the network, the width of the cliques and the number of states of the random variables in the cliques. In many cases exact inference must be performed in real time.

Almost all recent processors are designed to process simultaneous threads to achieve higher performance than single core processors. Typical examples of multicore processors available today include AMD Opteron and Intel Xeon. While chip multi-processing has been devised to deliver increased performance, an important challenge is to exploit the available parallelism. Prior work has shown that system performance is sensitive to thread scheduling in simultaneous multithreaded (SMT) architectures [6]. To maximize the potential of such multicore processors, users must understand both the algorithmic and architectural aspects to design efficient scheduling solutions.

In this paper, we study parallelization of evidence propagation on state-of-the-art multicore processors. We exploit both structural parallelism and data parallelism to improve the performance of evidence propagation. We achieved speedup of 7.4 using 8 cores on state-of-the-art platforms. This speedup is much higher compared with the baseline methods e.g. OpenMP based implementation. The proposed method can be extended for online scheduling of directed acyclic graph (DAG) structured computations.

The paper is organized as follows: In Section 2, we discuss the background of evidence propagation. Section 3 introduces related work. In Section 4, we present junction tree rerooting. Section 5 defines computation tasks for evidence propagation. Section 6 presents our collaborative scheduler for multicore processors. Experimental results are shown in Section 7. Section 8 concludes the paper.

2 Background

A *Bayesian network* is a probabilistic graphical model that exploits conditional independence to represent compactly a joint distribution. Figure 1 (a) shows a sample Bayesian network, where each node represents a random variable. The edges indicate the probabilistic dependence relationships between two random variables. Notice that these edges can *not* form directed cycles. Thus, the structure of a Bayesian network is a *directed acyclic graph* (DAG), denoted $\mathbb{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{A_1, A_2, \ldots, A_n\}$ is the node set and $\mathcal{E}$ is the edge set. Each random variable in the Bayesian network has a *conditional probability table* $P(A_j|pa(A_j))$, where $pa(A_j)$ is the parents of A_j. Given the Bayesian network, a joint distribution is given by $P(\mathcal{V}) = \prod_{j=1}^{n} P(A_j|pa(A_j))$, where $A_j \in \mathcal{V}$ [1].

The *evidence* in a Bayesian network is the variables that have been instantiated, e.g. $\mathrm{E} = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \ldots, n\}$, where A_{e_i} is a variable and a_{e_i} is the instantiated value. Evidence can be propagated to other variables in the Bayesian network using Bayes' Theorem. Propagating the evidence throughout a Bayesian network is called *inference*, which can be *exact* or *approximate*. Exact inference is proven to be NP hard [5]. The computational complexity of exact inference increases dramatically with the size of the Bayesian network and the number of states of the random variables.

Traditional exact inference using Bayes' theorem fails for networks with undirected cycles [1]. Most inference methods for networks with undirected cycles convert a network to a cycle-free hypergraph called a *junction tree*. We illustrate a junction tree converted from the Bayesian network (Figure 1 (a)) in Figure 1 (b), where all undirected cycles in are eliminated. Each vertex in Figure 1 (b) contains multiple random variables from the Bayesian network. For the sake of exploring evidence propagation in a junction tree, we use the following notations to formulate a junction tree. A junction tree is defined as $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$, where $\mathbb{T}$ represents a tree and $\hat{\mathbb{P}}$ denotes the parameter of the tree. Each vertex $\mathcal{C}_i$, known as a clique of J, is a set of random variables. Assuming $\mathcal{C}_i$ and $\mathcal{C}_j$ are adjacent, the *separator* between them is defined as $\mathcal{C}_i \cap \mathcal{C}_j$. $\hat{\mathbb{P}}$ is a set of *potential tables*. The potential table of $\mathcal{C}_i$, denoted $\psi_{\mathcal{C}_i}$, can be viewed as the joint distribution of the random variables in $\mathcal{C}_i$. For a clique with w variables, each having r states, the number of entries in $\mathcal{C}_i$ is r^w.

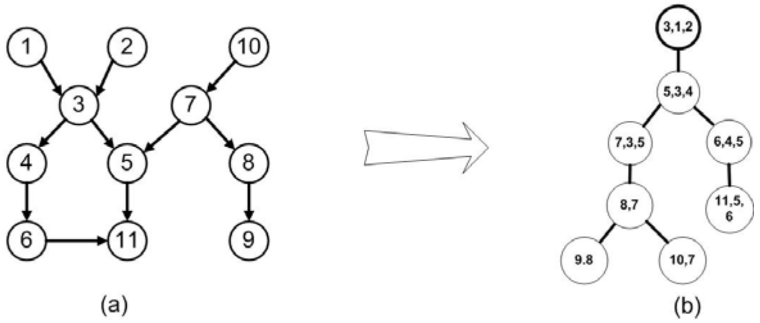


Fig. 1. (a) A sample Bayesian network and (b) corresponding junction tree

In a junction tree, exact inference proceeds as follows: Assuming evidence is $\mathrm{E} = \{A_i = a\}$ and $A_i \in \mathcal{C}_\mathcal{Y}$, E is *absorbed* at $\mathcal{C}_\mathcal{Y}$ by instantiating the variable A_i and renormalizing the remaining variables of the clique. The evidence is then propagated from $\mathcal{C}_\mathcal{Y}$ to any adjacent cliques $\mathcal{C}_\mathcal{X}$. Let $\psi^*_\mathcal{Y}$ denote the potential table of $\mathcal{C}_\mathcal{Y}$ after E is absorbed, and $\psi_\mathcal{X}$ the potential table of $\mathcal{C}_\mathcal{X}$. Mathematically, evidence propagation is represented as [1]:

$$\psi^*_\mathcal{S} = \sum_{\mathcal{Y}\backslash\mathcal{S}} \psi^*_\mathcal{Y}, \quad \psi^*_\mathcal{X} = \psi_\mathcal{X} \frac{\psi^*_\mathcal{S}}{\psi_\mathcal{S}} \tag{1}$$

where $\mathcal{S}$ is a separator between cliques $\mathcal{X}$ and $\mathcal{Y}$; $\psi_\mathcal{S}$ ($\psi^*_\mathcal{S}$) denotes the original (updated) potential table of $\mathcal{S}$; $\psi^*_\mathcal{X}$ is the updated potential table of $\mathcal{C}_\mathcal{X}$.

3 Related Work

There are several works on parallel exact inference, such as Pennock [5], Kozlov and Singh [7] and Szolovits. However, some of those methods, such as [7], are dependent upon the structure of the Bayesian network. The performance is adversely affected if the structure of the input Bayesian network is changed. Our method can be used for Bayesian networks and junction trees with various structures. Some other methods, such as [5], exhibit limited performance for multiple evidence inputs. The performance of our method does not depend on the number of evidence cliques. In [8], the authors discuss the structure conversion of Bayesian networks, which is different from evidence propagation addressed in this paper. In [9], the node level primitives are parallelized using message passing on distributed memory platforms. The optimization proposed in [9] is not applicable in this paper, since the multicore platforms have shared memory. However, the idea of parallelization of node level primitives is adapted by our scheduler to partition large tasks. A junction tree decomposition method is provided in [10] to partition junction trees for distributed memory platforms. This method reduces communication between processors by duplicating some cliques. We do not apply junction tree decomposition on our multicore platforms, because the clique duplication consumes memory that is shared by all the cores. A centralized scheduler for exact inference is introduced in [11], which is implemented on Cell BE, a heterogeneous multicore processor with a PowerPC element and 8 computing elements. However, the multicore platforms studied in this paper are homogeneous, and the number of cores is small. Using a separate core for centralized scheduling leads to performance loss. We deviate from the above approaches and explore collaborative task scheduling techniques for exact inference.

4 Junction Tree Rerooting for Minimizing Critical Path

A junction tree can be rerooted at any clique [5]. Consider rerooting a junction tree at clique $\mathcal{C}$. Let α be a preorder walk of the underlying undirected tree, starting from $\mathcal{C}$. Then, α encodes the desired new edge directions, i.e. an edge in the rerooted tree points from $\mathcal{C}_{\alpha_i}$ to $\mathcal{C}_{\alpha_j}$ if and only if $\alpha_i < \alpha_j$. In the rerooting procedure, we check the edges in the given junction tree and reverse any edges inconsistent with α. The result is a new junction tree rooted at $\mathcal{C}$, with the same underlying undirected topology as the original tree.

Rerooting a junction tree can lead to acceleration of evidence propagation on parallel computing systems. Let $P(\mathcal{C}_i, \mathcal{C}_j) = \mathcal{C}_i, \mathcal{C}_{i_1}, \mathcal{C}_{i_2}, ..., \mathcal{C}_j$ denote a path from $\mathcal{C}_i$ to $\mathcal{C}_j$ in a junction tree, and $L_{(\mathcal{C}_i,\mathcal{C}_j)}$ denote the weight of path $P(\mathcal{C}_i, \mathcal{C}_j)$. Given clique width $w_{\mathcal{C}_t}$ and clique degree k_t for a clique $\mathcal{C}_t \in P(\mathcal{C}_i, \mathcal{C}_j)$, the weight of the path $L_{(\mathcal{C}_i,\mathcal{C}_j)}$ is defined as:

$$L_{(\mathcal{C}_i,\mathcal{C}_j)} = \sum_{\mathcal{C}_t \in P(\mathcal{C}_r,\mathcal{C}_j)} k_t w_{\mathcal{C}_t} \prod_{l=1}^{w_{\mathcal{C}_t}} r_l \tag{2}$$

The *critical path* (CP) of a junction tree is defined as the longest weighted path of the junction tree. Give a junction tree $\mathbb{J}$, the weight of a critical path, denoted L_{CP}, is given by $L_{CP} = \max_{\mathcal{C}_j \in \mathbb{J}} L_{(\mathcal{C}_r, \mathcal{C}_j)}$, where $\mathcal{C}_r$ is the root. Notice that evidence propagation in a critical path takes at least as much time as that in other paths. Thus, among the rerooted junction trees, the one with the minimum critical path leads to the best performance on parallel computing platforms.

A straightforward approach to find the optimal rerooted tree is as follows: First, reroot the junction tree at each clique. Then, for each rerooted tree, calculate the weight of the critical path. Finally, select the rerooted tree corresponding to the minimum weight of the critical path. Given the number of cliques N and maximum clique width $w_{\mathcal{C}}$, the serial computational complexity of the above procedure is $O(N^2 w_{\mathcal{C}})$.

We present an efficient rerooting method (see Algorithm 1) to minimize the critical path, which is based on the following lemma:

Lemma 1. Suppose that $P(\mathcal{C}_x, \mathcal{C}_y)$ is the longest weighted path from a leaf clique $\mathcal{C}_x$ to another leaf clique $\mathcal{C}_y$ in a given junction tree, and $L_{(\mathcal{C}_r,\mathcal{C}_x)} \geq L_{(\mathcal{C}_r,\mathcal{C}_y)}$, where $\mathcal{C}_r$ is the root. Then, $P(\mathcal{C}_r, \mathcal{C}_x)$ is a critical path in the given junction tree.

Proof sketch. Assume a critical path is $P(\mathcal{C}_r, \mathcal{C}_z)$, $\mathcal{C}_z \neq \mathcal{C}_x$. Let $P(\mathcal{C}_r, \mathcal{C}_{b1})$ denote the longest common path between $P(\mathcal{C}_r, \mathcal{C}_x)$ and $P(\mathcal{C}_r, \mathcal{C}_y)$, and $P(\mathcal{C}_r, \mathcal{C}_{b2})$ the longest common path between $P(\mathcal{C}_r, \mathcal{C}_x)$ and $P(\mathcal{C}_r, \mathcal{C}_z)$. Without loss of generality, assume $\mathcal{C}_{b2} \in P(\mathcal{C}_r, \mathcal{C}_{b1})$. Since $P(\mathcal{C}_r, \mathcal{C}_z)$ is a critical path, we have $L_{(\mathcal{C}_r,\mathcal{C}_z)} \geq L_{(\mathcal{C}_r,\mathcal{C}_x)}$. Note that $L_{(\mathcal{C}_r,\mathcal{C}_z)} = L_{(\mathcal{C}_r,\mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2},\mathcal{C}_z)}$ and $L_{(\mathcal{C}_r,\mathcal{C}_x)} = L_{(\mathcal{C}_r,\mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2},\mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1},\mathcal{C}_x)}$. Therefore, $L_{(\mathcal{C}_{b2},\mathcal{C}_z)} \geq L_{(\mathcal{C}_{b2},\mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1},\mathcal{C}_x)} > L_{(\mathcal{C}_{b1},\mathcal{C}_x)}$. Thus, we can find path $P(\mathcal{C}_z, \mathcal{C}_y) = P(\mathcal{C}_z, \mathcal{C}_{b2}) P(\mathcal{C}_{b2}, \mathcal{C}_{b1}) P(\mathcal{C}_{b1}, \mathcal{C}_y)$ which leads to:

$$\begin{aligned} L_{(\mathcal{C}_z,\mathcal{C}_y)} &= L_{(\mathcal{C}_z,\mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2},\mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1},\mathcal{C}_y)} > L_{(\mathcal{C}_{b1},\mathcal{C}_x)} + L_{(\mathcal{C}_{b2},\mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1},\mathcal{C}_y)} \\ &> L_{(\mathcal{C}_x,\mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1},\mathcal{C}_y)} = L_{(\mathcal{C}_x,\mathcal{C}_y)} \end{aligned} \tag{3}$$

The above inequality contradicts the assumption that $P(\mathcal{C}_x, \mathcal{C}_y)$ is a longest weighted path in the given junction tree. □

According to Lemma 1, the new root can be found once we identify the longest weighted path between two leaves in the given junction tree. We introduce a tuple $\langle v_i, p_i, q_i \rangle$ for each clique $\mathcal{C}_i$ to find the longest weighted path (Lines 1-6 Algorithm 1), where v_i records the complexity of a critical path of the subtree rooted at $\mathcal{C}_i$; p_i and q_i represent $\mathcal{C}_{p_i}$ and $\mathcal{C}_{p_i}$, respectively, which are two children of $\mathcal{C}_i$. If $\mathcal{C}_i$ has no child, p_i and q_i are empty.

The path from $\mathcal{C}_{p_i}$ to some leaf clique in the subtree rooted at $\mathcal{C}_i$ is the *longest* weighted path among all paths from a child of $\mathcal{C}_i$ to a leaf clique, while the path from $\mathcal{C}_{q_i}$ to a certain leaf clique in the subtree rooted at $\mathcal{C}_i$ is the *second longest* weighted path. The two paths are concatenated at $\mathcal{C}_i$ and form a leaf-to-leaf path in the original junction tree. In Lines 3 and 4, $\arg_j \max(v_j)$ stands for the value of the given argument (parameter) j for which the value of the given expression v_j

Algorithm 1. Root selection for minimizing critical path

```
Input: Junction tree J
Output: New root $\mathcal{C}_r$
1: initialize a tuple $\langle v_i, p_i, q_i \rangle = \langle k_i w_{\mathcal{C}_i} \prod_{j=1}^{w_{\mathcal{C}_i}} r_j, 0, 0 \rangle$ for each $\mathcal{C}_i$ in J
2: for $i = N$ downto 1 do
3:     $p_i = \arg_j \max(v_j), \forall pa(\mathcal{C}_j) = \mathcal{C}_i$
4:     $q_i = \arg_j \max(v_j), \forall pa(\mathcal{C}_j) = \mathcal{C}_i$ and $j \neq p_i$
5:     $v_i = v_i + v_{p_i}$
6: end for
7: select $\mathcal{C}_m$ where $m = \arg_i \max(v_i + v_{q_i}), \forall i$
8: initialize path $P = \{\mathcal{C}_m\}$; $i = m$
9: while $\mathcal{C}_i$ is not a leaf clique do
10:     $i = p_i$; $P = \{\mathcal{C}_i\} \cup P$
11: end while
12: $P = P \cup \mathcal{C}_{q_m}$; $i = m$
13: while $\mathcal{C}_i$ is not a leaf node do
14:     $i = p_i$; $P = P \cup \{\mathcal{C}_i\}$
15: end while
16: denote $\mathcal{C}_x$ and $\mathcal{C}_y$ the two end cliques of path $P$
17: select new root $\mathcal{C}_r = \arg_i \min |L_{(\mathcal{C}_x,\mathcal{C}_i)} - L_{(\mathcal{C}_i,\mathcal{C}_y)}| \; \forall \mathcal{C}_i \in P(\mathcal{C}_x, \mathcal{C}_y)$
```

attains its maximum value. In Line 7, we detect a clique $\mathcal{C}_m$ on the longest weighted path and identify the path in Lines 8-15 accordingly. The new root is then selected in Line 17.

We briefly analyze the serial complexity of Algorithm 1. Line 1 takes $w_{\mathcal{C}}N$ time for initialization, where $w_{\mathcal{C}}$ is clique width and N is the number of cliques. The loop in Line 2 has N iterations and both Lines 3 and 4 take $O(k)$ time, where k is the maximum number of children of a clique. Line 7 takes $O(N)$ time, as do Lines 8-15, since a path consists of at most N cliques. Lines 16-17 can be completed in $O(N)$ time. Since $k < w_{\mathcal{C}}$, the serial complexity of Algorithm 1 is $O(w_{\mathcal{C}}N)$, compared to $O(w_{\mathcal{C}}N^2)$, the complexity of the straightforward approach.

5 Task Definition and Dependency Graph Construction

5.1 Task Definition

Evidence propagation consists of a series of computations called node level primitives. There are four types of node level primitives: *marginalization*, *extension*, *multiplication* and *division* [9]. In this paper, we define a *task* as the computation of a node level primitive. The input to each task is one or two potential tables, depending on the specific primitive type. The output is an updated potential table. The details of the primitives are discussed in [9]. We illustrate the tasks related to clique $\mathcal{C}$ in Figure 2 (b). Each number in brackets corresponds to a task of which the primitive type is given in Figure 2 (c). The dashed dashed arrows in Figure 2 (b) illustrate whether the task works on the same potential table or between two potential tables. The edge in Figure 2 (c) represent precedence order of the execution of the tasks.

A property of the primitives is that the potential table of a clique can be partitioned into independent activities and processed in parallel. The results from each activity are combined (for extension, multiplication and division) or added (for marginalization) to obtain the final output. This property is utilized in Section 6.

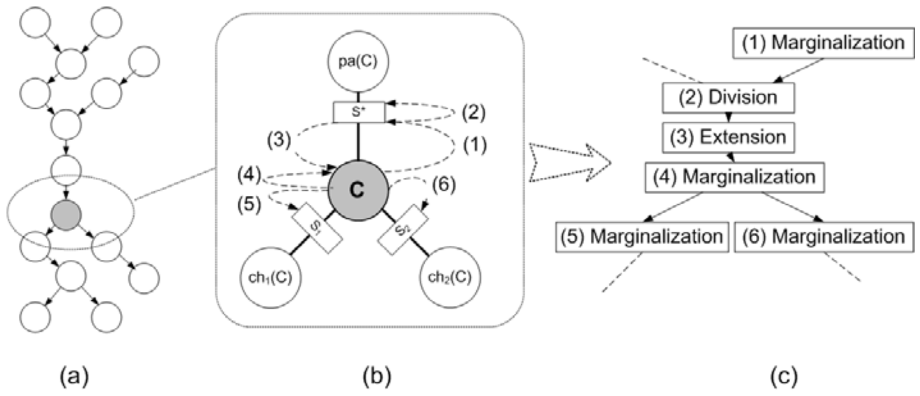


Fig. 2. (a) Clique updating graph; (b) Primitives used to update a clique; (c) Local task dependency graph with respect to the clique in (b)

5.2 Dependency Graph Construction

Given an arbitrary junction tree, we reroot it according to Section 4. The resulting tree is denoted $\mathbb{J}$. We construct a *task dependency graph G* from $\mathbb{J}$ to describe the precedence constraints among the tasks. The task dependency graph is created in the following two steps:

First, we construct a *clique updating graph* to describe the coarse grained dependency relationship between cliques in $\mathbb{J}$. In exact inference, $\mathbb{J}$ is updated twice [1]: (1) evidence is propagated from leaf cliques to the root; (2) evidence is then propagated from the root to the leaf cliques. Thus, the clique updating graph has two symmetric parts. In the first part, each clique depends on all its children in $\mathbb{J}$. In the second part, each clique depends on its parent in $\mathbb{J}$. Figure 2 (a) shows a sample clique updating graph from the junction tree given in Figure 1 (b).

Second, based on the clique updating graph, we construct *task dependency graph G* to describe the fine grained dependency relationship between the tasks defined in Section 5.1. The tasks related to a clique $\mathcal{C}$ are shown in Figure 2 (b). Considering the precedence order of the tasks, we obtain a small DAG called a *local task dependency graph* (see Figure 2 (c)). Replacing each clique in Figure 2 (a) with its corresponding local task dependency graph, we obtain the task dependency graph G for junction tree $\mathbb{J}$.

6 Collaborative Scheduling

We propose a *collaborative scheduler* to allocate the tasks in the task dependency graph G to the cores. We assume that there are P cores in a system. The framework of the scheduler is shown in Figure 3. The *global task list* (GL) in Figure 3 stores the tasks from the task dependency graph. Each entry of the list stores a task and the related data, such as the task size, the task dependency degree, and the links to its succeeding tasks. Initially, the *dependency degree* of a task is the number of incoming edges of the task in G. Only the tasks with dependency degree equal to 0 can be processed. The global task list is shared by all the threads, so any thread can fetch a task, append new tasks, or decrease the dependency degree of tasks. Before an entry of the list is accessed by a thread, all the data in the entry must be protected by a lock to avoid concurrent write.

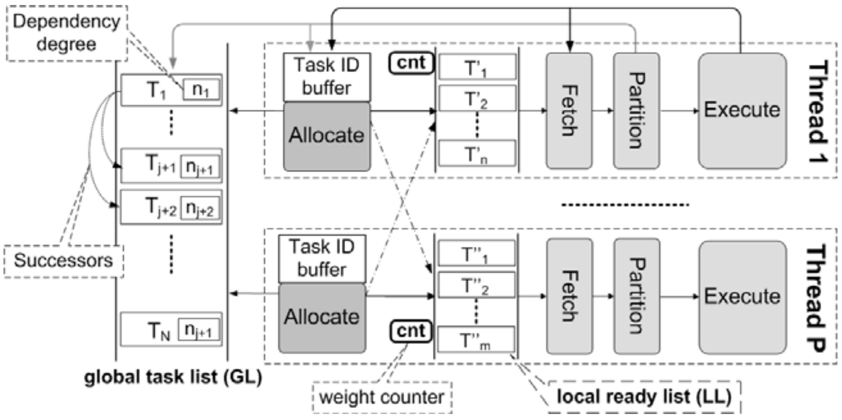


Fig. 3. Components of the collaborative scheduler

Every thread has an *Allocate module* which is in charge of decreasing task dependency degrees and allocating tasks to the threads. The module only decreases the dependency degree of the tasks if their predecessors appear in the *Task ID buffer* (see Figure 3). The task ID corresponds to the offset of the task in the GL, so the module can find the task given the ID in $O(1)$ time. If the dependency degree of a task becomes 0 after the decrease operation, the module allocates it to a thread with the aim of load balancing across threads. Various heuristics can be used to balance the workload. In this paper, we allocate a task to the thread with the smallest workload.

Each thread has a *local ready list* (LL) to store the tasks allocated to the thread. All the tasks in a LL are processed by the same thread. However, since the tasks in the LL can be allocated by all the Allocate modules, the LLs are actually global. Thus, locks are used to prevent concurrent write to LL. Each LL has a *weight counter* to record the workload of the tasks in the LL. Once a new task is inserted to (fetched from) the list, the workload of the task is added to (subtracted from) the weight counter.

The *Fetch module* takes tasks from the LL in the same thread. Heuristics can be used to select tasks from the LL. In this paper, we use a straightforward method where the task at the head of the LL is fetched.

The *Partition module* checks the workload of the fetched task. The tasks with heavy workload are partitioned for load balancing. As we discussed in Section 5.1, a property of the primitives is that the potential table of a clique can be partitioned easily. Thus, a task T can be partitioned to subtasks $\hat{T}_1, \hat{T}_2, \cdots, \hat{T}_n$, each processing a part of the potential table related to task T. Each subtask inherits the parents of T in the task dependency graph G. However, we let $\hat{T}_n$ be the successor of $\hat{T}_1, \cdots, \hat{T}_{n-1}$, and only $\hat{T}_n$ inherits the successor of T. Therefore, the results from the subtasks can be concatenated or added by $\hat{T}_n$. The Partition module preserves the structure of G, except replacing T by the subtasks. The module replaces T in the GL with $\hat{T}_n$, and appends other subtasks to the GL. $\hat{T}_1$ is sent to the Execute module and $\hat{T}_2, \cdots, \hat{T}_{n-1}$ are evenly distributed to local lists, so that these subtasks can be executed by several threads.

Each thread also has a local *Execute module* where the primitive related to a task is performed. Once the primitive is completed, the Execute module sends the ID of the task to the Task ID buffer, so that the Allocate module can accordingly decrease the dependency degree of the successors of the task. The Execute module also signals the Fetch module to take the next task, if any, from LL.

The collaborative scheduling algorithm is shown in Algorithm 2. We use the following notations in the algorithm: GL is the global list. LL_i is the local ready list in Thread i. d_T and w_T denote the dependency degree and the weight of task T, respectively. W_i is the total weight of the tasks in LL_i. δ is the threshold of the size of potential table. Any potential table larger than δ is partitioned. Line 1 in Algorithm 2 initializes the Task ID buffers. As shown in Line 3, the scheduler keeps on working until all tasks are processed. Lines 4-10 correspond to the Allocate module. Line 11 is the Fetch module. Lines 12-18 correspond to the Partition module and Execute Module.

Algorithm 2 achieves load balancing by two means: First, the Allocate module ensures that the new tasks are allocated to the threads where the total workload of the tasks in its LL is the lowest. Second, the Partition module guarantees that each single large task can be processed in parallel.

7 Experiments

We conducted experiments on two state-of-the-art homogeneous multicore processor systems: Intel Xeon quadcore and AMD Opteron quadcore system. The

Algorithm 2. Collaborative Task Scheduling

```
1: $\forall\, T$ s.t. $d_T = 0$, evenly distribute the ID of $T$ to Task ID buffers
2: for Thread $i$ ($i = 1 \ldots P$) in parallel do
3:   while GL$\cup$LL$_i \neq \emptyset$ do
4:     for $T \in \{$ successors of tasks in the $i$-th Task ID buffer $\}$ do
5:       $d_T = d_T - 1$
6:       if $d_T = 0$ then
7:         allocate $T$ to $LL_j$ where $j = \arg_t \min(W_t), t = 1 \cdots P$
8:         $W_k = W_k + w_T$
9:       end if
10:    end for
11:    fetch task $T'$ from $LL_i$
12:    if the size of potential table $\psi_{T'} > \delta$ then
13:      partition $T'$ into subtasks $\hat{T}'_1, \hat{T}'_2, \cdots, \hat{T}'_n$ s.t. $\psi_{\hat{T}'_j} \leq \delta,\ j = 1, \cdots, n$
14:      replace $T'$ in GL with $\hat{T}'_n$, and allocate $\hat{T}'_1, \cdots, \hat{T}'_{n-1}$ to local lists
15:      execute $\hat{T}'_1$ and place the task ID of $\hat{T}'_1$ into the $i$-th Task ID buffer
16:    else
17:      execute $T'$ and place the task ID of $T'$ into the $i$-th Task ID buffer
18:    end if
19:  end while
20: end for
```

former contained two Intel Xeon x86_64 E5335 processors, each having four cores. The processors ran at 2.00 GHz with 4 MB cache and 16 GB memory. The operating system was Red Hat Enterprise Linux WS release 4 (Nahant Update 7). We installed GCC version 4.1.2 compiler and Intel C/C++ Compiler (ICC) version 10.0 with streaming SIMD extensions 3 (SSE 3), also known as Prescott New Instructions (PNI). The latter platform had two AMD Opteron 2347 quadcore processors, running at 1.9 GHz. The system had 16 GB DDR2 memory and the operating system was Red Hat Linux CentOS version 5. We also used GCC 4.1.2 compiler on the AMD platform.

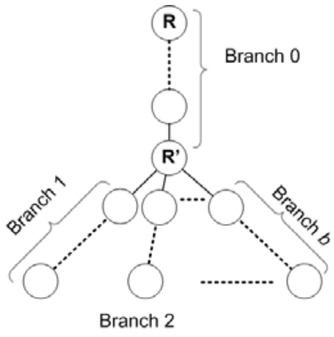


Fig. 4. Junction tree template for evaluating rerooting algorithm

To evaluate the performance of the junction tree rerooting method shown in Algorithm 1, we generated four junction trees using the template shown in Figure 4. The template in Figure 4 is a tree with $b + 1$ branches. R is the root. Using $b = 1, 2, 4, 8$, we obtained four junction trees. Every junction tree had 512 cliques, each consisting of 15 binary variables. Thus, the serial complexity of each Branch is approximately equal. Using Algorithm 1, clique R' became the new root after rerooting. For each junction tree, we performed evidence

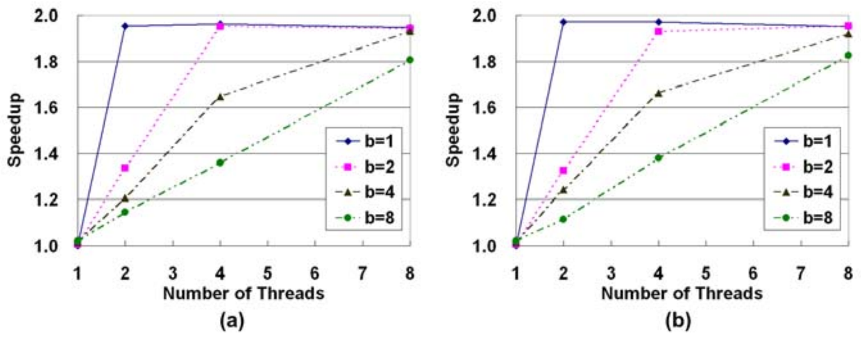


Fig. 5. Speedup due to rerooting on (a) Intel Xeon and (b) AMD Opteron. $(b+1)$ is the total number of branches in the junction tree.

propagation on both the original tree and the rerooted tree, using various number of cores. We disabled task partitioning, which provided parallelism at fine grained level.

The results are shown in Figure 5. The speedup in Figure 5 was defined as $Sp = t_R/t_{R'}$, where t_R $(t_{R'})$ is the execution time of evidence propagation in the original (rerooted) junction tree. According to Section 4, we know that when clique R is the root, Branch 0 plus Branch 1 is a critical path. When R' is the root, only Branch 0 is the critical path. Thus, the maximum speedup is 2 for the four junction trees, if the number of concurrent threads P is larger than b. When $P < b$, Sp was less than 2, since some cliques without precedence constraint can not be processed in parallel. From the results in Figure 5, we can see that the rerooted tree led to speedup around 1.9, when 8 cores were used. In addition, the maximum speedup was achieved using more threads as b increases. These observations matched the analysis above. Notice that some speedup curves fell slightly when 8 concurrent threads were used. This was caused by the overheads such as the lock contention.

We also observed that, compared with the time for evidence propagation, the percentage of overall execution time spent on junction tree rerooting was very small. Rerooting the junction tree with 512 cliques took 24 microseconds on the AMD Opteron quadcore system, compared to 2.8×10^7 microseconds for the overall execution time. Thus, although Algorithm 1 was not parallelized, it causes negligible overhead in parallel evidence propagation.

We generated junction trees of various sizes to analyze and evaluate the performance of the proposed evidence propagation method. The junction trees were generated using Bayes Net Toolbox [12]. The first junction tree (Junction tree 1) had 512 cliques and the average clique width was 20. The average degree for each clique was 4. All random variables were binary. The second junction tree (Junction tree 2) had 256 cliques and the average clique width was 15. The number of states of random variables was 3 and the average clique degree was 4. The third junction tree (Junction tree 3) had 128 cliques. The clique width was 10 and the number of states of random variables was 3. The average clique degree was 2. All the three junction trees were rerooted using Algorithm 1. In our experiments, we used double precision floating point numbers to represent the probabilities and potentials.

We performed exact inference with respect to the above three junction trees using Intel Open Source Probabilistic Network Library (PNL) [13]. The scalability of the results is shown in Figure 6. PNL is a full function, free, open source, graphical model library released under Berkeley Software Distribution (BSD) style license. PNL provides an implementation for junction tree inference with discrete parameters. The parallel version of PNL is also provided by Intel [13]. The results shown in Figure 6 were obtained on a IBM P655 multiprocessor system, where each processor runs at 1.5 GHz with 2 GB of memory. We can see from Figure 6 that, for all the three junction trees, the execution time increased when the number of processors was greater than 4.

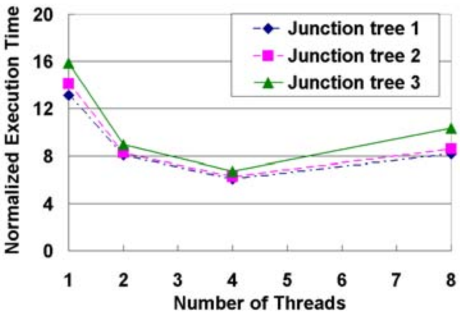


Fig. 6. Scalability of exact inference using PNL library for various junction trees

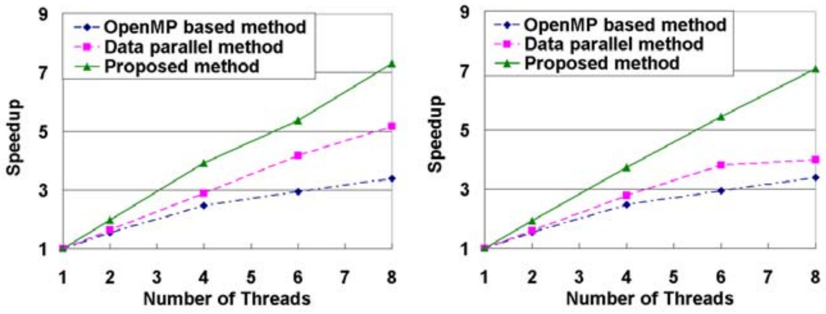


Fig. 7. Scalability of exact inference using various methods on (a) Intel Xeon and (b) AMD Opteron

We compared three parallel methods for evidence propagation on both Intel Xeon and AMD Opteron in Figure 7. The first two methods were the baselines. The first parallel method was the *OpenMP based method*, where the OpenMP intrinsic functions were used to parallelize the sequential code. We used ICC to compile the code on Xeon, while GCC was used on Opteron. The second method is called *data parallel method*, where we created multiple threads for each node level primitive. That is, the node level primitives were parallelized every time they were performed. The data parallel method is similar to the task partitioning mechanism in our collaborative scheduler, but the overheads were large. The third parallel method was the proposed method. Using Junction trees 1-3 introduced above, we conducted experiments on both the platforms. For all the three methods, we used level 3 optimization (`-O3`). The OpenMP based method also benefited from the SSE3 optimization (`-msse3`). We show the speedups in Figure 7. The results show that the proposed method exhibited linear speedup and was superior compared with the baseline methods on both the platforms. Performing the proposed method on 8 cores, we observed speedup of 7.4 on Intel Xeon and 7.1 on AMD Opteron. Compared to the OpenMP based method, our approach achieved speedup of 2.1 when 8 cores were used. Compared to the data parallel method, we achieved speedup of 1.8 on AMD Opteron.

To show the load balance we achieved and the overhead of the collaborative scheduler, we measured the computation time for each thread. In our context, the computation time for a thread is the total time taken by the thread to perform node level primitives. Thus, the time taken to fetch tasks, allocate tasks and maintain the local ready list were not considered. The computation time reflects the workload for each thread. We show the results in Figure 8 (a), which were obtained on Opteron using Junction tree 1 defined above. We observed very similar results for Junction tree 2 and 3. Due to space constraints, we show results on Junction tree 1 only. We also calculated the ratio of the computation time over the total parallel execution time. This ratio illustrates the quality of the scheduler. From Figure 8 (b), we can see that, although the scheduling time increased a little as the number of threads increases, it was not exceeding 0.9% of the execution time for all the threads.

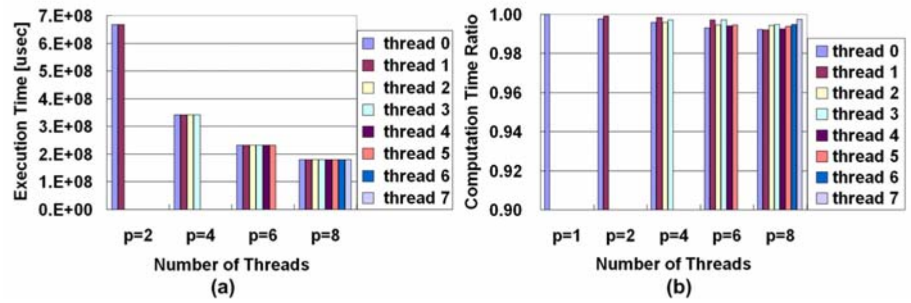


Fig. 8. (a) Load balance across the threads; (b) Computation time ratio for each thread

Finally, we modified parameters of Junction tree 1 to obtain a dozen junction trees. We applied the proposed method on these junction trees to observe the performance of our method in various situations. We varied the number of cliques N, clique width w_C, number of states r and average number of children k. We obtained almost linear speedup for all cases. From the results in Figure 9 (a), we observe that the speedups achieved in the experiments with various values for N were all above 7. All of them exhibited linear speedups. In Figure 9 (b) and (c),

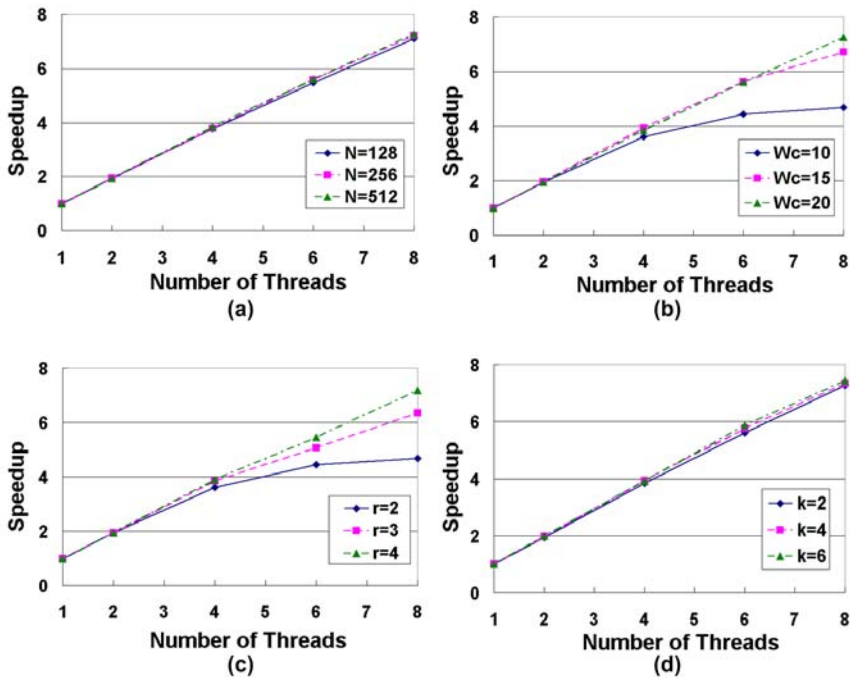


Fig. 9. Speedups of exact inference on multicore systems with respect to various junction tree parameters

all results showed linear speedup except the ones with $w_{\mathcal{C}} = 10$ and $r = 2$. The reason was that the size of the potential table was small. For $w_{\mathcal{C}} = 10$ and $r = 2$, the potential table had 1024 entries, about 1/1000 of the number of entries in a potential table with $w_{\mathcal{C}} = 20$. However, since N and the junction tree structure were the same, the scheduling requires approximately the same time for junction tree with small potential tables. Thus, the overheads became relatively large. In Figure 9 (d), all the results had similar performance when k was varied. All of them achieved speedups of more than 7 using 8 cores.

8 Conclusions

We presented an efficient rerooting algorithm and a collaborative scheduling algorithm for parallel evidence propagation. The proposed method exploited both task and data parallelism in evidence propagation. Thus, even though one of the levels can not provide enough parallelism, the proposed method still achieves speedup on parallel platforms. Our implementation achieved 7.4× speedup using 8 cores. This speedup is much higher compared with the baseline methods, such as the OpenMP based implementation. In the future, we plan to investigate the overheads in the collaborative scheduler and further improve its performance.

As more cores are integrated into a single chip, some overheads such as lock contention will increase dramatically. We intend to improve the design of the collaborative scheduler to reduce such overheads, so that the scheduler can be used for a class of DAG structured computations in the many-core era.

References

1. Lauritzen, S.L., Spiegelhalter, D.J.: Local computation with probabilities and graphical structures and their application to expert systems. J. Royal Statistical Society B 50, 157–224 (1988)
2. Heckerman, D.: Bayesian networks for data mining. Data Mining and Knowledge Discovery (1997)
3. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice-Hall, Englewood Cliffs (2002)
4. Segal, E., Taskar, B., Gasch, A., Friedman, N., Koller, D.: Rich probabilistic models for gene expression. In: 9th International Conference on Intelligent Systems for Molecular Biology, pp. 243–252 (2001)
5. Pennock, D.: Logarithmic time parallel Bayesian inference. In: Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence, pp. 431–438 (1998)
6. De Vuyst, M., Kumar, R., Tullsen, D.: Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–6 (2006)
7. Kozlov, A.V., Singh, J.P.: A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In: Supercomputing, pp. 320–329 (1994)
8. Xia, Y., Prasanna, V.K.: Parallel exact inference. In: Proceedings of the Parallel Computing, pp. 185–192 (2007)
9. Xia, Y., Prasanna, V.K.: Node level primitives for parallel exact inference. In: Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing, pp. 221–228 (2007)
10. Xia, Y., Prasanna, V.K.: Junction tree decomposition for parallel exact inference. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–12 (2008)
11. Xia, Y., Prasanna, V.K.: Parallel exact inference on the cell broadband engine processor. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12 (2008)
12. Murphy, K.: http://www.cs.ubc.ca/~murphyk/software/bnt/bnt.html
13. Intel Open Source Probabilistic Networks Library, http://www.intel.com/technology/computing/pnl/

Parallelization of Temperature Distribution Simulations for Semiconductor and Polymer Composite Material on Distributed Memory Architecture

Norma Alias[1,⋆], Roziha Darwis[2], Noriza Satam[1], and Mohamed Othman[2]

[1] Universiti Teknologi Malaysia, Skudai, Johor, Malaysia
[2] Universiti Putra Malaysia, Serdang, Selangor, Malaysia
norma@ibnusina.utm.my, {roziha.darwis,norizasatam}@gmail.com, mothman@fsktm.upm.edu.my

Abstract. The implementations of parallel algorithms in solving partial differential equations (PDEs) for heat transfer problems are based on the high performance computing using distributed memory architecture. In this paper, the parallel algorithms are exploited finite difference method in solving multidimensional heat transfer problem for semiconductor components and polymer composite materials. Parallel Virtual Machine (PVM) and C language based on Linux operating system are the platform to run the parallel algorithms. This research focused on Red-Black Gauss Seidel (RBGS) iterative method. Parallel performance evaluations in terms of speedup, efficiency, effectiveness, temporal performance and communication cost are analyzed.

Keywords: Parallel Virtual Machine (PVM), Red-Black Gauss Seidel (RBGS), Parallel Performance evaluations.

1 Introduction

The methodology to predict temperature is based on mathematical simulation that focuses on parabolic and elliptic type of PDEs. The experiment will covers both semiconductor components and tire tread problem. The discretization of PDEs will generate linear system of equations dealing with sequential computational of large sparse matrices. This high computational complexity can be solved efficiently by transforming the algorithms from sequential to parallel. Huge executions of numerical method using several numbers of processors will coined the numerical analysis in terms of convergence, accuracy and parallel performance evaluations. The objective of this paper is to predict the temperature behavior of semiconductor component and polymer composite material based on mathematical modelling using parallelization.

⋆ The authors acknowledge Ibnu Sina Institute, UTM and Ministry of Science, Technology and Innovation Malaysia for financial support, under research vot 79217 and 79219.

2 Mathematical Modelling on PDEs

This paper focuses on parabolic equation to visualize temperature behavior of tire treads and semiconductor wire problem. Meanwhile, will be used elliptic equation to predict multilayer full chip systems.

2.1 Polymer Composite Material

Temperature behavior for polymer composite material is focused on phase change simulations for two types of parabolic equations associate with two phases simulations. Phases simulations involves heating and cooling process.

Mathematical model under consideration is one-dimensional parabolic equation based on two-phase Stefan problem [4] involving the appearance of a new phase change. Prediction of liquid and solid simulation is given by,

$$\rho_l C_l \frac{\partial U_l}{\partial t} = \lambda_l \frac{\partial^2 U_l}{\partial x^2} + \rho_l U_l, \quad 0 < x < X(t)\ . \tag{1}$$

$$\rho_s C_s \frac{\partial U_s}{\partial t} = \lambda_s \frac{\partial^2 U_s}{\partial x^2} + \rho_s U_s, \quad x > X(t)\ . \tag{2}$$

where ρ, C and λ represents the density, heat capacity and thermal conductivity of rubber. Subscript l and s indicate liquid and solid state of rubber and U represent the temperature profile $U(x,t)$.

2.2 Semiconductor of Multilayered Full Chip

In semiconductor manufacturing industries, accurate temperature behavior prediction is an important issues in quality improvement of product. Mathematical simulation of the prediction involves PDEs with elliptic type and has ability to deal with large sparse matrices. The two-dimensional elliptic equation is governed by the following Poisson's equation [1],

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)\ . \tag{3}$$

2.3 Semiconductor of Wires

The three-dimensional heat conduction in semiconductor wires problem involving the time dependent of parabolic PDEs type. The equation that contains partial derivatives on the dependent variables are given by [5],

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} + h(x,y,t)\ . \tag{4}$$

where U is interpreted as the time dependent of temperature profile per unit volume respect to the $x-, y-$ and $z-$ axis. The heat conduction in semiconductor wire having a constant cross section of the volume and it is depend on x, y, z and t if the wire is not uniformly heating.

3 Parallel Algorithm

Parallel algorithm in parallelizing (1), (2), (3) and (4) is Red Black Gauss-Seidel(RBGS) Iteration Method. This method contains 2 subdomain, Ω^R and Ω^H [6]. The calculation involves,

i. Grid calculation at Ω^R

$$u^{k+1} = \frac{1}{a}(f_i - cu_{i-1}^{k+1} - bu_{i+1}^{k}), i = 1, 3, 5, ..., m \; . \tag{5}$$

ii. Grid calculation at Ω^H

$$u^{k+1} = \frac{1}{a}(f_i - cu_{i-1}^{k+1} - bu_{i+1}^{k+1}), i = 2, 4, 6, ..., m-1 \; . \tag{6}$$

The parallelization of RBGS can be directly implemented on distributed memory architecture. This is due to the data independence between two subdomains. Parallel implementation will be evaluated as more processors added to the cluster of distributed memory architecture.

Parallelized RBGS method converged faster compared to sequential RBGS. Domain decomposition technique allows array division among local processors and this will minimize communication. The structure of data has to be decomposed where given set of ranges assigned to particular processors must be physically sent to those processors during execution. The result must be sent back to processor that responsible for coordinating the final result. The procedure for parallel algorithm as show in Fig. 1 is pseudocode and communication between master and workers in distributed memory architecture.

4 Parallel Performance Evaluations

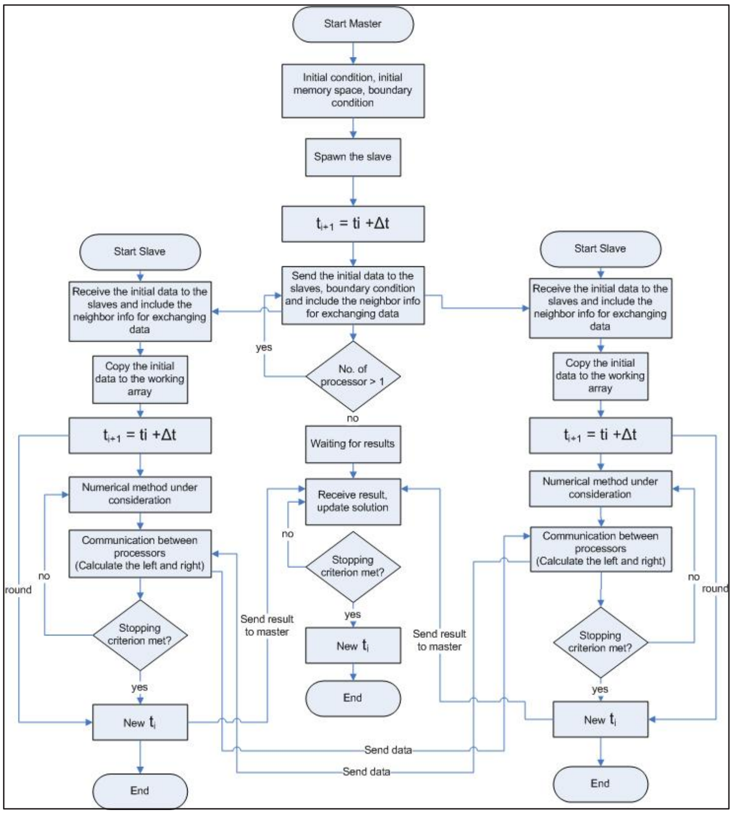
The analysis of parallel performance evaluation be is done in terms of execution time, speedup, efficiency, effectiveness, temporal performance and computational complexity. The parallel performance evaluation metric are as described below,

a. Speedup,$S_p = \frac{T_1}{T_p}$
b. Efficiency,$C_p = \frac{S_p}{p}$

where, T_1 is the execution time using one processor, and T_p represents the execution time on p processors.

The important factors affecting performance are communication cost, computational and communication ratio. Communication cost will depends on many factors includes network structure and contention [3]. Parallel execution time, t_{para} divided into two parts namely, computational time, t_{comp} and communication time, t_{comp}. t_{comp} is time consumed to compute the arithmetic operations such as multiplication and addition. As all processors doing the operation at the same speed, calculation for t_{comp} is depends upon the size of message. Communication cost comes from two major phases in sending a message: start-up

```
Set number of iterations
**Master's session**
Invoke workers
Send variables and initial data to workers
For round : = 1 increase 1 until number of iterations do
For timestep :=1 until convergence criterion met do
**Worker's session**
Receive variables and initia   l data from master
Calculations of power
For i = 1 increase until matrix size do
For j = 1 increase until matrix size do
If odd, do
$u^{k+1} = \frac{1}{a}(f_i - cu^{k+1}_{i-1} - bu^{k}_{i+1})$;
End loop i
Communication between workers (left and right)
For i = 1 incre ase until matrix size do
For j = 1 increase until matrix size do If even, do
$u^{k+1} = \frac{1}{a}(f_i - cu^{k+1}_{i-1} - bu^{k}_{i+1})$;
End loop j
End loop i
Check convergence
Communication between workers (left and right)
Send convergence analysis to master
```



Start Master
Initial condition, initial memory space, boundary condition
Spawn the slave
t_{i+1} = ti +Δt
Send the initial data to the slaves, boundary condition and include the neighbor info for exchanging data
yes
No. of processor > 1
no
Waiting for results
Receive result, update solution
no
Stopping criterion met?
yes
New t_i
End
Start Slave
Receive the initial data to the slaves and include the neighbor info for exchanging data
Copy the initial data to the working array
t_{i+1} = ti +Δt
Numerical method under consideration
Communication between processors (Calculate the left and right)
Stopping criterion met?
no
round
yes
New t_i
End
Send result to master
Send data
Send data

Fig. 1. Pseudocode and parallel algorithm

and data transmission phase [3]. Total time to send K units of data for a given system can be written as,

$$t_{comm} = t_{startup} + Kt_{data} + t_{idle}$$

where t_{comm} is time needed to communicate a K bytes message and $t_{startup}$ is sometimes referred as network latency time. $t_{startup}$ is also referred as time to send a message with no data. It includes time to pack message at source and unpack the message at intended destination as well as to start a point-to-point communication. t_{data} is time to transmit units of information. $t_{startup}$ and t_{data} assumed as constants and measured in bits/ sec. t_{idle} is the time for message latency and time to wait for all processors to complete the tasks.

4.1 Parallel Performance of 1D Phase Change Simulation

Parallel performance measurement for one-dimensional problem of tire treads is shown in Fig. 2, by using matrices size are $m = 100 \times 100$ with time execution is $225.88241\mu s$, iteration is 150, Δx=1.0000E_{-2}, Δt=1.0000E_{-2} and ϵ=1.0E_{-15}.

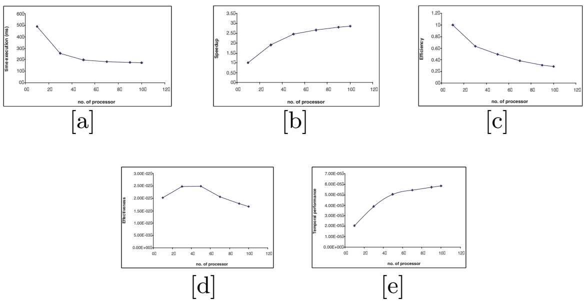


Fig. 2. Parallel performance evaluation for tire treads in terms of (a)Execution time (b) Speedup (c) Efficiency (d) Effectiveness (e)Temporal performance

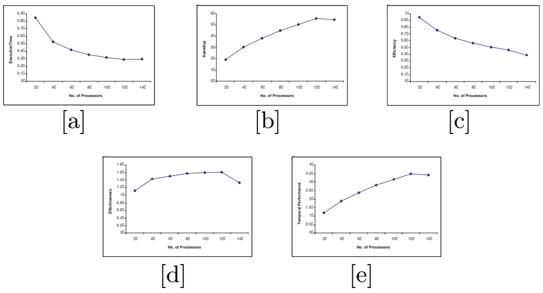


Fig. 3. Parallel performance evaluation for multilayer full chip in terms of (a) Execution time (b) Speedup (c) Efficiency (d) Effectiveness (e) Temporal performance

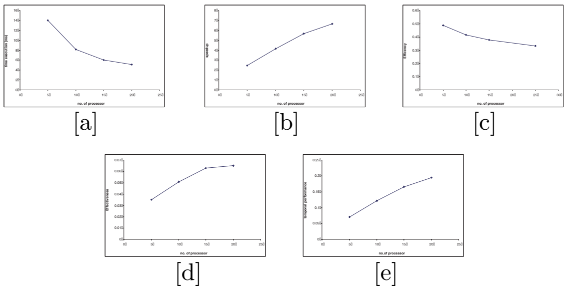


Fig. 4. Parallel performance evaluation for semiconductor wires in terms of (a) Execution time (b) Speedup (c) Efficiency (d) Effectiveness (e) Temporal performance

4.2 Parallel Performance of 2D Poisson Equation

Parallel performance evaluation for two-dimensional of multilayered full chip is as shown in Fig.3, by using matrices size are $m = 400 \times 400$ with time execution is $84.7263\mu s$, iteration is 200, Δx=1.0000, Δy=1.0000, Δt=1.0000 and ϵ=1.0E_{-3}.

4.3 Parallel Performance of 3D Parabolic Equation

Fig. 4 shows the parallel performance evaluation for three-dimensional problem of semiconductor wires. by using matrices size are $m = 100 \times 100 \times 100$ with time execution is $282.9480\mu s$, iteration is 760, Δx=1.0000E_{-2}, Δy=1.0000E_{-2}, Δz=1.0000E_{-2}, Δt=6.6667E_{-1} and ϵ=1.0E_{-9}.

Fig. 2 to 4 show the parallel performance evaluations in terms of execution time, speedup, efficiency, effectiveness and temporal performance. Figure 1(a), 2(a) and 3(a) show the time execution decreased upon increment of processor number. In measuring efficiency of parallel algorithm, it is normal if the execution results in significant decrement. This is due to two possible reasons such as efficiency decrease as the number of processors increase and efficiency increase as the size of matrices increase.

The drop of effectiveness shows that the problem under consideration will effectively being solved using 10 to 20 numbers of processors. Improvement of temporal performance evaluations are as illustrated in Figure 1(e), 2(e) and 3(e).

Numerical analysis for three-dimensional semiconductor wires is shown in Table 1.

Table 1. Computational and Communication cost for different size of matrices

m	100×100 × 100					140×140 × 140				
p	exec	comp	ratio	comm	idle	exec	comp	ratio	comm	idle
5	99.9	58.59	1.42	41.31	11.87	193	111.8	1.38	81.25	32.11
10	73.01	29.29	0.67	43.71	14.27	136.2	55.89	0.7	80.34	21.21
15	69.11	19.73	0.4	49.27	19.84	121	36.66	0.43	84.29	35.15
20	63.99	15.15	0.31	48.85	19.42	110.7	27.15	0.32	83.55	34.41

5 Conclusion

Temperature visualization based on PDEs will result in accurate prediction and suits well in distributed memory architecture environment. The parallel RBGS method is proven to be efficient and effective method in solving multidimensional problems that involved semiconductor and polymer composite material. Additionally, distributed memory architecture that supports high computational complexity and communication cost has significant ability in solving parabolic and elliptic types of problem.

References

1. Zhan, Y., Sapatnekar, S.S.: A High Efficiency Full-Chip Thermal Simulation Algorithm. In: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design, pp. 635–638 (2005)
2. Juma, M., Bafrnec, M.: Experimental Determination of Rubber Curing Reaction Heat Using the Transient Heat Conduction Equation. Chemical Papers 58(1), 29–32 (2004)
3. Wilkinson, B., Allen, M.: Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice Hall, Upper Saddle River (1999)
4. Solomon, A., Alexiades, V., Wilson, D.G.: The initial velocity of the emerging free boundary in a two-phase Stefan Problem with imposed flux. Siam J. Math. Anal. 18(5) (September 1987)
5. Gourlay, R., Mcguire, G.R.: J. Inst. Math. Appl. 7, 216 (1971)
6. University of Cambridge: Lecture Notes (Partial Differential Equation), http://www.damtp.cam.ac.uk/lab/people/sd/lectures/nummeth98/pdes.htm

Implementation of a Non-bonded Interaction Calculation Algorithm for the Cell Architecture

Eduard Fomin and Nikolay Alemasov

Institute of Cytology and Genetics, Siberian Branch of the Rusian Academy of Science,
10 Lavrentiev Ave., 630090, Novosibirsk, Russia
{fomin,alemasov}@bionet.nsc.ru

Abstract. Calculation of non-bonded interactions takes up to 80% of the total execution time of a molecular dynamics program. It can be accelerated by porting the algorithm to the Cell architecture. A simple method of such porting has been applied to the MOLKERN program, which simulates the structure and dynamics of biomolecular models. A 32-fold speedup was achieved for calculation of short-range non-bonded interactions, and threefold, for long-range Coulomb interactions. The overall program speedup proved to be more than 4.

Keywords: Molecular simulation programs, pairwise non-bonded interactions, Cell processors.

1 Introduction

Programs simulating the structure and dynamics of biomolecular models constructed on the force-field approximation are broadly used in molecular biology. This approximation assumes that the potential energy function of atom interaction can be presented with practically sufficient accuracy, e.g., for the AMBER force field [1], as:

$$U = \sum U_b + \sum U_a + \sum U_d + \sum U_{coul} + \sum U_{vdw},$$

where U_b terms are energies of valence bonds; U_a, of valence angles; U_d, of torsion angles; and U_{coul} and U_{vdw}, of the Coulomb and Van der Waals interactions. The first three sums in U are defined only for locally bound atoms; therefore, the total number of terms in the $\sum U_b$, $\sum U_a$ and $\sum U_d$ sums is in direct proportion with the number of atoms in the assemblage O(N). The last two sums are determined for all atom pairs of the assemblage, and the total number of terms in U_{coul} and U_{vdw} is in proportion with $O(N^2)$. These terms form the bottleneck in molecular dynamics programs. For example, the GROMACS program was tested by simulation of the small (35 aa) villin headpiece protein in a cell containing 3000 water molecules, and the calculation under discussion constituted 83% of the runtime [2].

There are three basic strategies for mapping the algorithm onto multiple-instruction multiple-data (MIMD) machines [3]: (1) replicated data, i.e. each processor keeps a copy of all data in its memory, but it works only with the part of the data that is assigned to it; (2) particle decomposition, i.e., assignment of particles to processors using their indices; and (3) domain decomposition, i.e., assignment of particles to processors according to their spatial location. These mapping strategies can be helpful to process large molecular systems containing a great number of atoms, but the long-time simulation of such systems depends only on the speed of the individual processors of the MIMD machine. Problems of this sort can be best solved by using hybrid MIMD/SIMD architectures, such as the STI Cell processor [4].

Cell/BE [5] has nine cores. One of them is a PowerPC Processor Element (PPE), a processor of the PowerPC architecture. The other eight cores are Synergistic Processor Elements (SPEs). Their architecture differs from that of PPE, and they are used as main calculating devices. Each SPE has its own local 256 K storage (LS). It stores both the code and data. Each SPE has a set of 128 128-bit registers and a large SIMD instruction set. Owing to such architecture features as the heterogeneous multi-core structure, parallelism at both coarse (across SPEs) and fine (within each SPE) granularities, high data transfer bandwidth (200GB/s on chip and 25.6 GB/s off chip for the 3.2GHz processor), and explicit local memory control through DMA, the Cell/BE processor demonstrates tremendous superiority over others.

MOLKERN [6] has been designed for tasks involving simulation of the structure and dynamics of biomolecule models including hundreds of thousands of atoms. MOLKERN implements optimized algorithms with computational complexity no more than O(N log N). MOLKERN was written in C++ with the use of the libraries STL, BOOST, BLAS, and MPI and the OpenMP. The original MOLKERN version was implemented for x86 processors. The purpose of the present work is to port the MOLKERN program to the Cell architecture and optimize the most time-consuming functions, calculating the energies and forces created by all pairwise interactions.

2 Methods

2.1 Algorithm

A simple algorithm of non-bonded interaction calculation has the computational complexity $O(N^2)$. It is suitable for calculating small assemblages of few hundreds of atoms. The following pseudocode describes the calculation:

```
foreach (atom_pair in atom_pair_list) {
        {par1, par2} = get_params (atom1, atom2);
        force = calculate_force(par1, par2);
        write(force, atom1, atom2);
}
```

The algorithm runs across the list of atom pairs, gets the relevant data for either atom of a pair from the memory, calculates the force acting between the atoms, and puts the result to the memory.

In actual calculations, the algorithm is modified to reduce the computational complexity. All atom pairs are classified into short- and long-range, depending on the distance between the atoms. The boundary between them is conventionally defined as cutoff radius r_{cut}, chosen so that at $r>r_{cut}$ potentials should monotonously decrease. In this way, pairwise interactions can be summarized at short distances in the coordinate space according to the above pseudocode, and the number of such pairs varies directly as O(N). Transformation in the reciprocal k-space is used for calculating forces involving long-range pairs. The pseudocode for this calculation (e.g., in PPPM [7]) is as follows:

```
foreach (atom) Q.insert(atom.charge, atom.X)
Q_ = dfft_forward(Q);
foreach (k in Q_) G_[k] = Q_[k]*coul_fourier(|k|)*k;
G = dfft_backward(G_);
foreach (atom) interpolation(G, atom.X);
```

This algorithm performs:

- calculation of the smooth charge distribution Q on the grid,
- direct Fourier transform of Q,
- calculation of the k-image of the Coulomb potential gradient G_, totaled over all k-vectors, on the grid,
- inverse Fourier transform of the k-image of gradient G_ to the coordinate space, and
- linear interpolation of the gradient determined at grid nodes to atom locations to obtain forces.

The computational complexity of the algorithm, equaling O(N log N), is determined by the most time-consuming function, related to the Fourier transform.

2.2 Implementation

MOLKERN follows the common approach to pairwise interaction calculation. It generates the neighbor list in the range $r < r_{cut}$, directly summarizes van der Waals and short-range Coulomb interactions for these neighbors, and calculates the long-range Coulomb part of the potential by the PPPM method for $r > r_{cut}$. MOLKERN profiling indicates that the greatest part of the calculation time (up to 92%) is taken by calculation of non-bonded interactions, including:

- direct summing of van der Waals and short-range Coulomb interactions for pairs in the neighbor list (45–58% of the total time at $r_{cut} = 1$ nm) and
- calculation of the long-range Coulomb interactions by the PPPM method (34–46% of the total time at the grid size 0.2 nm).

The Cell version of MOLKERN is implemented for a server with two PowerXCell8i processors, which had 2 PPEs and 16 SPEs.

Calculation of Short-range Non-bonded Interactions

The sequential version of the program uses an algorithm based on the above pseudocode. The simplest method of parallel processing of this algorithm involves the OpenMP interface [8]. In contract to x86, where our use of OpenMP was successful and yielded 45% gain per additional core [9], the performance of the algorithm on the

Cell platform was tens of times worse. We analyzed this fact and found that the program demanded too much data transfer between the main memory and SPE LS. Such transfer between SPE and PPE cannot be controlled through OpenMP.

Thus, to optimize memory calls, the calculation of short-range pairwise interactions was split into three functions performing the following tasks:

1. get() - reading all data from the memory to a data block.
2. calculate() - calculation and summation of energies and forces for pairs of block.
3. put() - transfer of the calculation results from the data block to the memory.

The calculation was performed block-by-block, and each block was processed by a separate SPE, whereas data transfer between the block and the memory was performed by four threads of two PPEs. This data transfer was implemented with OpenMP. The block-by-block approach is characterized by natural parallelization and unnecessity of synchronization of subtasks executed in different SPEs, as the blocks are independent of each other. The pseudocode for a single block is:

```
foreach (atom in atom_pair_list[select_atom]) {
          pars = get_params (atom);
          write(block, pars); // PPE + OpenMP
}
block.calculate(force); // SPE only
write(force, select_atom); // PPE
```

The procedure of data loading into blocks made use of the ordering of pairs naturally provided by the nearest neighbor search algorithm. To process a particular atom, blocks were loaded only with data for its neighbors, as far as the block size permitted. It allowed the SPE to perform partial summation of forces produced by the neighbors falling into the block. With the presence of several blocks storing data for neighbors of the atom under consideration, the final summations were performed by the PPE during data transfer to memory. In this approach, the block size affects the distribution of efforts on force summation between the SPE and PPE. Analysis of the dependence of program performance on block size allowed the optimal value of 32K to be chosen. With this size, for the majority of blocks all neighbors of a certain atom fall into one block; therefore, additional summation of forces by the PPEs is not needed. Larger block sizes do not provide better speed. Smaller blocks reduce the performance because the work on force summation is redistributed from SPEs to PPEs.

To implement the function, we wrote the code for thread initialization and data transfer between the PPE and SPEs. It was no larger than 300 lines. Also, slight modifications related to involvement of SIMD extensions were applied to codes performing calculations within blocks. The IBM SDK libspe2 and MASS SIMD libraries [10] were used in the implementation.

Calculation of Long-range Non-bonded Interactions

Profiling of the code calculating long-range non-bonded interactions showed that Fourier transform was its bottleneck (about ¾ of the code execution time). A simple way for porting this code utilizes the FFTW library [11], which supports this platform starting from version 3.2. A drawback of this porting strategy is that the speedup does not exceed 4 according to rough estimation.

2.3 Tests

The performance of the Cell version of the algorithm was tested on 3 proteins: 1AIE, 1AF2 and 1GC1 in a cell containing 2008, 5346 and 36873 water molecules, respectively. The comparison was carried out with reference to the sequential MOLKERN version, performed entirely by PPE. The results were obtained by averaging the time of execution of short-range interaction calculation over the first five iterations of geometry optimization. The calculation of non-bonded interactions involved the potentials: 6–12 van der Waals, the short-range Coulomb component $\mathrm{erfc}(\sqrt{\pi} * r / r_{cut}) / r$, and the long-range Coulomb component $\mathrm{erf}(\sqrt{\pi} * r / r_{cut}) / r$.

3 Results and Discussion

Figure 1 presents a line graph illustrating the speedup of calculation of short-range non-bonded interactions by the MOLKERN depending on the number of SPEs employed. It shows the results for various versions of the SPE code differing in use of double or single precision and in using SIMD extensions. A log scale is used. Dispersion of the results in various test calculations is shown.

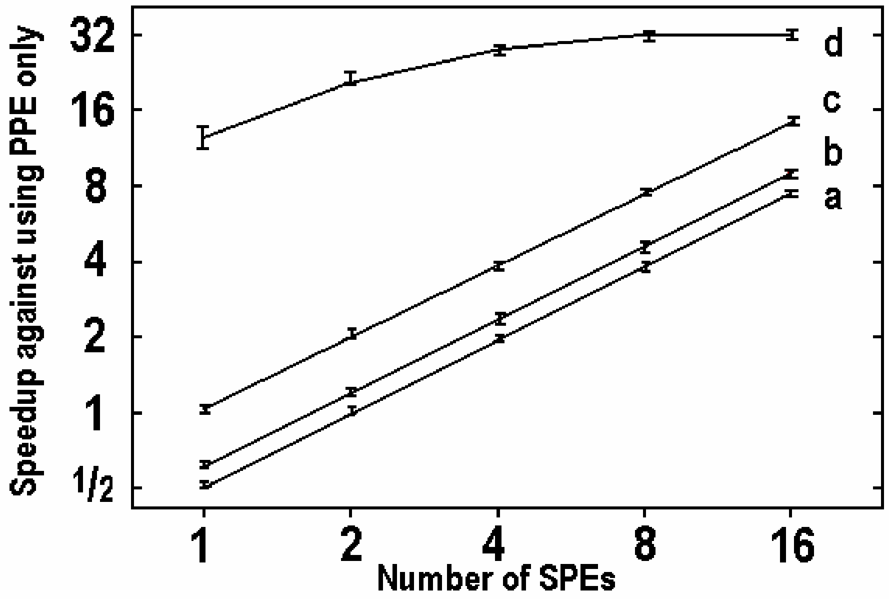


Fig. 1. Dependence of the speedup of short-range interaction calculation on the number of SPEs employed. The results for various versions of the SPE code are shown: (a) double-precise without SIMD extensions; (b) double-precise with SIMD extensions; (c) single-precise without SIMD extensions; and (d) single-precise with SIMD extensions.

For (a), (b), and (c), the SPEs were completely loaded, and the speed of calculations linearly increased with the number of SPEs. The maximum speedup of all these codes, equaling 14.6, was achieved in case (c) with all the 16 SPEs. When SIMD extensions were used with single precision (d), the maximum speedup equaling 31.6 was achieved with 8 SPEs. The dependence of speedup on the number of SPEs was not linear, which indicated that the load was improperly distributed between PPEs and SPEs. SPE code was executed so fast that the four threads in two PPEs do not manage to supply data. Thus, Cell has a great potential for further calculation acceleration by utilizing SPEs inactive in this version. The relative error for the code executed by SPEs in comparison with the sequential version was no more than 10^{-6}.

The improvement of the performance of the function calculating the long-range Coulomb potential in our version was limited by our porting strategy to 3.2. A great acceleration of this portion of the algorithm is expected with total porting of the PPPM method to SPEs to avoid the delay related to data preparation for FFT.

To sum up, the MOLKERN program was ported to the Cell architecture. The program code is executed by PPE, whereas SPEs perform only the calculation of non-bonded interactions. Almost 32-fold speedup was achieved for calculation of short-range non-bonded interactions, and threefold, for long-range Coulomb interactions. The overall performance of the program increased significantly, more than fourfold.

Acknowledgments. This work was supported by Interdisciplinary SB RAS Basic Research Integration projects 26 "Mathematical models, numerical methods, and parallel algorithms for large tasks from the Siberian Branch of the Russian Academy of Sciences and their implementation on multiprocessor supercomputers" and 113 "Development of computation methods, algorithms, hardware, and software for parallel simulation of natural processes". We are grateful to the T-platforms Company (http://www.t-platforms.ru/) for providing access to the PowerXCell8i server.

References

1. Ponder, J.W., Case, D.A.: Force fields for protein simulations. Adv. Prot. Chem. 66, 27–85 (2003)
2. GROMACS: Fast, Free and Flexible MD. Benchmarks, `http://www.gromacs.org`
3. Griebel, M., Knapek, S., Zumbusch, G.: Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications. Springer, Heidelberg (2007)
4. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P.: The Potential of the Cell Processor for Scientific Computing. In: Proceedings of the 3rd conference on Computing frontiers, Ischia, Italy, pp. 9–20 (2006)
5. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell multiprocessor. IBM Journal of Research and Development 49(4/5), 589–604 (2005)
6. Fomin, E.S., Alemasov, N.A., Chirtsov, A.S., Fomin, A.E.: MOLKERN: A library of software components for molecular modeling programs. Biophysics 51(suppl. 1), 110–112 (2006)
7. Hockney, R., Eastwood, J.: Computer simulation using particles. McGraw-Hill, New York (1981)

8. The OpenMP API specification for parallel programming, `http://www.openmp.org`
9. Alemasov, N.A., Fomin, E.S.: OPENMP+MPI parallel implementation of the MOLKERN molecular modeling software package. In: Proceedings of the 6th International Conference on Bioinformatics of Genome Regulation and Structure, p. 24 (2008)
10. IBM SDK for Multicore acceleration v 3.0, `http://www.ibm.com/developerworks`
11. Frigo, M., Johnson, S.G.: FFTW on the Cell Processor, `http://www.fftw.org/cell/index.html`

A Parallel 3D Code for Simulation of Self-gravitating Gas-Dust Systems*

Sergei Kireev

ICMMG SB RAS, Novosibirsk, Russia
kireev@ssd.sscc.ru

Abstract. A parallel 3D code for simulation of galaxies and protoplanetary discs is developed. The model includes dust, gas, gravitation and friction between dust and gas. The kinetic equation for dust particles is solved by PIC method. Gas dynamics equations are solved by FLIC method. In parallel implementation a domain decomposition technique is used where each subdomain is processed by a group of processors. Results of parallelization efficiency are presented.

1 Introduction

Using numerical simulation for solution of many astrophysical problems imposes certain requirements on numerical models and their implementations. For example, doing research of matter movement in galaxy or planet formation in protoplanetary disc one has to consider several physical processes and observe phenomena on different spatial and temporal scales. A program for numerical simulation must be able to utilize a large computational power that is to be ready to run on supercomputers.

The paper considers a parallel 3D code for investigation of astrophysical objects such as galaxies and protoplanetary discs. A numerical model widely used for simulation of such objects includes dust and gas components and gravitational interaction. Other processes such as chemical reactions, electromagnetics, radiation are also took into account. Complexity of parallel implementation results from the necessity to combine different physical processes and to use different numerical methods in one implementation.

In recent years many astrophysical codes were developed, which solve N-body problem together with gas dynamics and self-gravitation [1,2,3,4,5,6]. They differ in methods used for solution of gas dynamics equations and methods of gravitational forces calculation. In this paper a PIC (particle-in-cell) method is used for simulation of dust component and a FLIC (fluid-in-cell) method is used for simulation of gas component. It is a natural extension of 2D model described in [7] to the 3D case.

* Supported by Subprogram 18-2 of RAS Presidium Program "Biosphere origin and evolution", Subprogram *П*-04 of RAS Presidium Program "Stars and galaxies origin and evolution", RFBR (grant 08-01-00615), SB RAS Program on Supercomputers, Grant of Rosobrazovanie, contract PH*П*.2.2.1.1.3653.

2 Mathematical Model

The mathematical model used in this paper includes collisionless Boltzman (Vlasov-Liouville) equation (1) for dust component, four gas dynamics equations (2) and Poisson equation (3) for self-gravitation.

$$\frac{\partial f}{\partial t} + \boldsymbol{u}\frac{\partial f}{\partial \boldsymbol{r}} + \boldsymbol{a}\frac{\partial f}{\partial \boldsymbol{u}} = 0, \tag{1}$$

$$\begin{gathered}
\frac{\partial \rho_g}{\partial t} + div(\rho_g \boldsymbol{v}) = 0,\\
\frac{\partial}{\partial t}(\rho_g \boldsymbol{v}) + div\left[(\rho_g \boldsymbol{v})\boldsymbol{v}\right] = -grad\, p - \rho_g grad \mathit{\Phi} + k_{fr}\rho_p\rho_g(\boldsymbol{u}-\boldsymbol{v}),\\
\frac{\partial}{\partial t}(\rho_g E) + div\left[(\rho_g E)\boldsymbol{v}\right] = -div(p\boldsymbol{v}) - \rho_g(grad\mathit{\Phi}, \boldsymbol{v}) + k_{fr}\rho_p\rho_g(\boldsymbol{u}-\boldsymbol{v}, \boldsymbol{u}),\\
\frac{\partial p}{\partial t} + div(p\boldsymbol{v}) = (\gamma - 1)\left[-p\, div\boldsymbol{v} + k_{fr}\rho_p\rho_g(\boldsymbol{u}-\boldsymbol{v})^2\right]
\end{gathered} \tag{2}$$

$$\Delta \mathit{\Phi}_2 = 4\pi\rho. \tag{3}$$

In (1–3) $f(t, \boldsymbol{r}, \boldsymbol{u})$ is time-dependent one-particle distribution function along coordinates $\boldsymbol{r}$ and velocities $\boldsymbol{u}$, $\boldsymbol{a} = \boldsymbol{F}_p + \boldsymbol{F}^p_{fr}$ is the acceleration of unit mass particle.

Gravitational potential $\mathit{\Phi}$ can be divided in two parts: $\mathit{\Phi} = \mathit{\Phi}_1 + \mathit{\Phi}_2$, where $\mathit{\Phi}_1$ is the outer potential (for example, potential of the central mass of galaxy or protoplanetary disc), and $\mathit{\Phi}_2$ is a self-consistent potential of the moving matter, satisfying the Poisson equation (3). $\rho = \rho_p + \rho_g$ is the aggregate density of dust and gas components, where ρ_g is a gas density and ρ_p is a dust density, which is determined by the equation $\rho_p = \int f(t, \boldsymbol{r}, \boldsymbol{u})d\boldsymbol{u}$. p is a gas pressure, $\boldsymbol{v}$ is a gas velocity, $E = \frac{T}{\gamma-1} + \frac{\boldsymbol{v}^2}{2}$ is energy of the gas, where the temperature T satisfies the equation of state: $p = \rho_g T$.

Gravitational forces affecting dust and gas are respectively $\boldsymbol{F}_p = -\rho_p \nabla \mathit{\Phi}$ and $\boldsymbol{F}_g = -\rho_g \nabla \mathit{\Phi}$. Besides, there is a friction force between dust and gas: $\boldsymbol{F}^p_{fr} = -\boldsymbol{F}^g_{fr} = k_{fr}\rho_p\rho_g(\boldsymbol{v}-\boldsymbol{u})$. The friction results in the gas heating: $Q_{fr} = k_{fr}\rho_p\rho_g(\boldsymbol{u}-\boldsymbol{v})^2$, where k_{fr} is the coefficient of friction.

All equations are in dimensionless form.

3 Numerical Methods

For discretization of 3D simulation domain a uniform Cartesizian grid is used. The same grid is applied for simulation of dust, gas and Poisson equation solution. Boltzman equation (1) is solved by particle-in-cell (particle-mesh) method [9,10]. It reduces the solution of 6D equation to the solution of 3D equations of movement of a large number of model particles. Gas dynamics equations (2) are solved by modified FLIC method [11]. The Poisson equation (3) is solved by 3D discrete Fourier transform. The input of the Poisson solver is a distribution of summary dust and gas density. The output is a distribution of gravitational potential. To improve the precision of calculations the region borders are moved afar from the center. So, there are two nested grids of different size – the smaller one for solving dust and gas equations and the bigger one for solving the Poisson equation.

The simulation algorithm includes the following operations performed on each time step:

1. Shift of model particles by gravitational and frictional forces.
2. Calculation of gas properties on the smaller grid.
3. Calculation of dust density distribution on the smaller grid.
4. Calculation of dust velocity distribution on the smaller grid.
5. Calculation of summary dust-gas density distribution on the smaller grid.
6. Solution of Poisson equation on the bigger grid.
7. Calculation of gravitational forces on the smaller grid.

Correctness of solving equations (1-3) for dust, gas and gravitation were separately verified on analytical solutions. Control over solution of the whole task is performed using conservation laws: mass, momentum and full energy.

4 Parallel Algorithm

Parallel implementation of the algorithm aims at solving large-scale problems in acceptable time. The difficulty of PIC-method parallelization is that one must deal with a fixed grid and free-moving particles. In the present implementation scaling by the size of the grid is achieved by domain decomposition and scaling by the number of particles – by processing particles of one subdomain by a group of processors [12]. The planes dividing the computation domain are orthogonal to the disc plane. In such a way we avoid initially non-uniform distribution of particles between processors.

Every subdomain is assigned to a group of processors. The grid values (dust density, dust velocities, gravitational potential and gravitational forces) of the whole subdomain are stored in all processors of the group and the particles of the subdomain are uniformly distributed between the processors. The initial division of processors into groups is performed according to initial distribution of particles.

To preserve uniform distribution of particles between processors a load balancing algorithm was implemented. This algorithm is a parallel implementation of the new idea based on dynamic redistribution of processors among subdomains according to particles distribution.

Each group of processors has a main processor which holds gas parameters (density, momentum, velocity, pressure and full energy) and performs the solution of gas dynamics equations. Explicit schemes are used in FLIC method, so it is parallelized in a natural way using domain decomposition technique [8]. To calculate friction forces the values of gas density and velocity from the main processor of each group should be broadcasted to the rest processes of the group.

When solving Poisson equation parallel Fourier transform is performed by the free FFTW library. The library itself defines the decomposition of the bigger grid. So, there are two grids of different size and differently distributed over the processors (Fig. 1). That is why we need to redistribute the grid values twice per time step: density values from the smaller to the bigger grid and gravitational potential values from the bigger to the smaller grid.

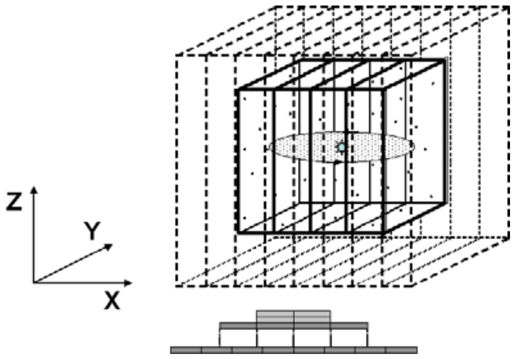


Fig. 1. Decomposition of bigger and smaller grids

The parallel algorithm has a number of parameters, such as number of groups and number of processors in each group. Their choice defines the efficiency of the algorithm. The following decisions were made:

– Only one processor in each group performs the gas calculations, so it is natural to set the number of groups to maximum.
– Initial distribution of processors among groups should be determined according to initial particles distribution. In general case the number of processors required in each group is unknown before particles are distributed. So, an algorithm for step-by-step distribution of particles and processors among groups was implemented. It eliminates the situation when one group has no processors to hold particles whereas processors in some other group have a plenty of free space. The final initial distribution of processors is obtained by a load balance algorithm.

There are also a number of constraints on parameter values. For example, the minimum number of groups is limited by the size of the cluster node memory. The maximum number of groups is limited by the size of the smaller grid along X axis. The maximum number of processors that could be used for Poisson equation solution is limited by the size of the larger grid along X axis. One other possible drawback of the algorithm is that it does not take into account a physical topology of the cluster.

The parallel program was implemented using Fortran and C languages. An MPI library is used for interprocess communications.

5 Simulation Results

For evaluation of parallel algorithm efficiency several test runs were made separately for dust and gas components with gravitation and for the compound self-gravitating gas-dust medium (Table 1). Initial substance distribution is a thin disc in the center of the simulation domain. Therefore after decomposition central subdomains contain many particles whereas subdomains near edges contain no particles. All runs were performed on cluster MVS100k in Joint Supercomputer Center in Moscow. Each cluster node contains two quad-core Xeon 3.0 GHz processors and not less then 4 GB of RAM. So, there are maximum 8 MPI processes per node.

Table 1. Task parameters for test runs

N	Component	Smaller grid size	Larger grid size	Number of particles	Size of data for one process run
(1)	dust	$200 \times 200 \times 200$	$400 \times 400 \times 400$	10^7	2.1 GB
(2)	dust	$300 \times 300 \times 300$	$600 \times 600 \times 600$	10^7	5.8 GB
(3)	dust	$200 \times 200 \times 200$	$400 \times 400 \times 400$	10^8	7.0 GB
(4)	gas	$200 \times 200 \times 200$	$400 \times 400 \times 400$	–	2.0 GB
(5)	gas	$250 \times 250 \times 250$	$500 \times 500 \times 500$	–	4.0 GB
(6)	dust + gas	$500 \times 500 \times 500$	$1000 \times 1000 \times 1000$	10^9	93.0 GB

Figure 2 presents computation time, speedup and parallelization efficiency for dust simulation with different sets of parameters. 100 time steps were performed. A number of groups was set to maximum in each case.

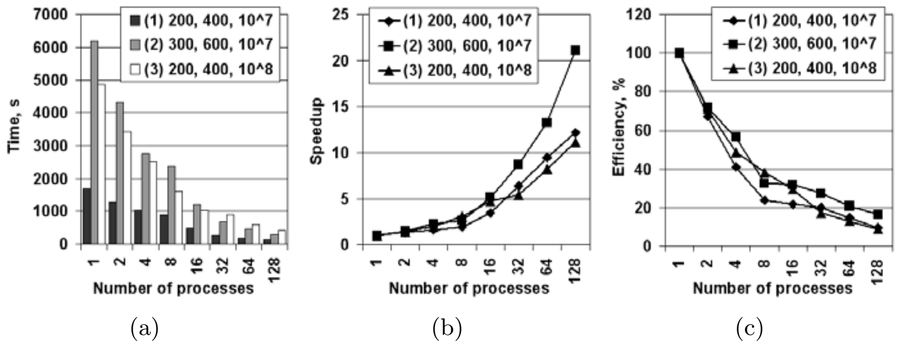


Fig. 2. Computation time (a), speedup (b) and efficiency (c) for calculation of dust component with three different sets of parameters using different number of processes

Results demonstrate a strong efficiency fall when processes reside in one node, and after 8 processes the scalability is acceptable. With increase of the mesh size the parallelization efficiency increases. But with increase of the number of particles the efficiency sometimes decreases for certain number of processes.

The following test runs were performed to obtain the optimal number of groups for these tasks (Fig. 3).

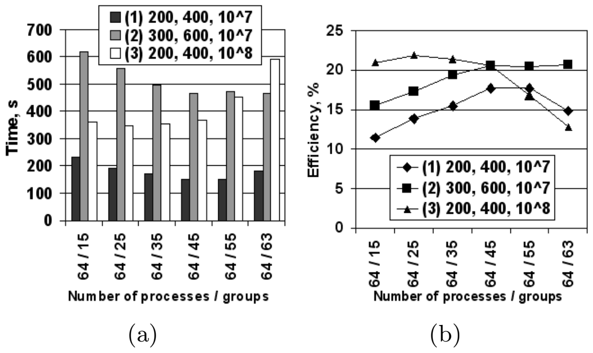


Fig. 3. Computation time (a) and efficiency (b) for calculation of dust component with three different sets of parameters using 64 processes and different number of groups

The results show that the task (2) with a larger grid works faster with a bigger number of groups. The task (3) with a larger number of particles works faster with a smaller number of groups, because more processes are put into the groups processing subdomains with large number of particles. So, depending on the prevalence of the grid size over the number of particles the optimal number of groups will move to one or other direction.

Figure (Fig. 4) presents computation time, speedup and parallelization efficiency for gas simulation with different sets of parameters. 100 time steps were performed. A number of groups was set to maximum in each case.

After 8 processes the scalability is good. The most strong efficiency fall took place from 1 to 8 processes in all tasks. The reason is a limit on memory bandwidth in a cluster node. For comparison the same tasks up to 8 processes were run using only one core in each node (Fig. 5). The results show that the efficiency fall is smaller when using processes in different nodes, especially for the task with large meshes. So, the poor memory architecture results in great performance degradation.

Figure 6 shows results for 100 time steps of gas-dust medium simulation using different number of processes and groups. The time of computations decreases with increase of the number of processes. But appropriate number of groups should be chosen depending on the parameters of the task as well as on characteristics of a cluster.

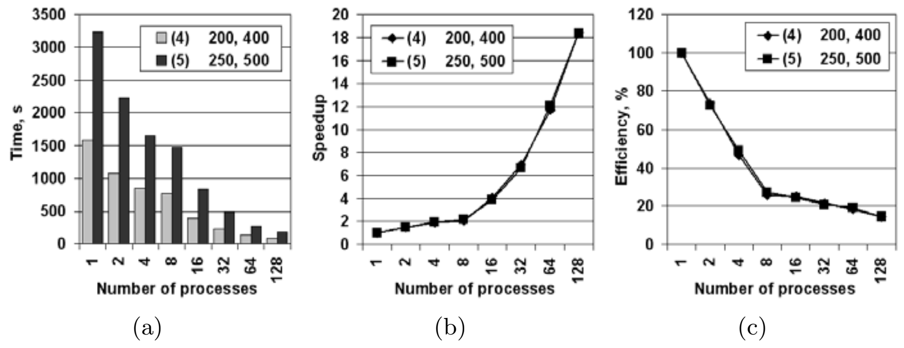


Fig. 4. Computation time (a), speedup (b) and efficiency (c) for calculation of gas component with two different sets of parameters using different number of processes

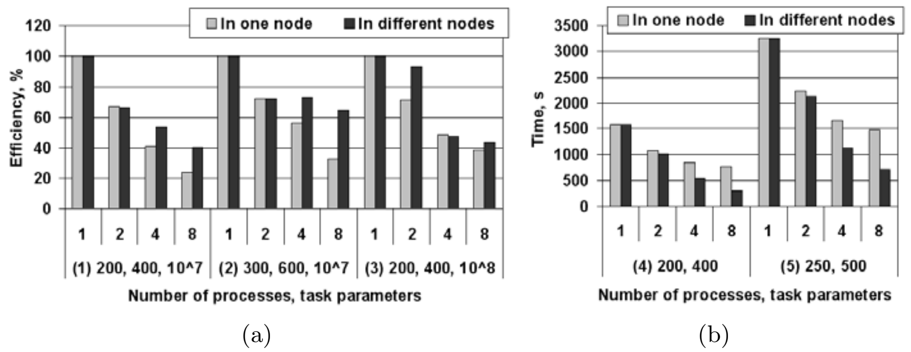


Fig. 5. Comparison of computation time and efficiency for dust (a) and gas (b) simulation using processes in one node or in different nodes

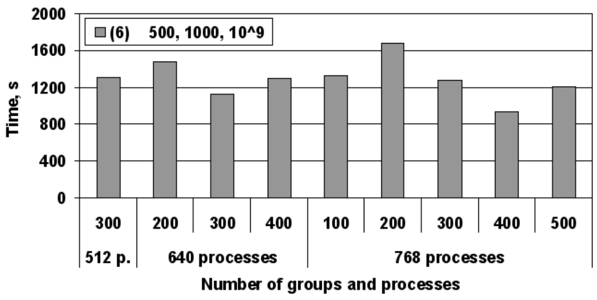


Fig. 6. Time of calculation for gas-dust self-gravitating medium simulation using different number of processes and groups

6 Conclusion

A parallel algorithm is developed and parallel code is implemented for 3D simulation of self-gravitating gas-dust systems. The code allows performing large-scale computations on a large number of processors in a reasonable time. Obtained scalability is acceptable. The main reason of parallelization efficiency decrease with increase of the number of processors is a memory bandwidth limit in a cluster node. This is a result of poor memory architecture. The presented implementation would show better results on processors with integrated memory controller, which eliminates the memory bottleneck. The other two reasons of efficiency decrease are the result of selected algorithm: a load imbalance due to a large decomposition granularity and a communication overhead.

Acknowledgements. The author would like to thank Dr. V.A. Vshivkov for scientific advising.

References

1. Miniati, F., Colella, P.: Block Structured Adaptive Mesh and Time Refinement for Hybrid, Hyperbolic + N-body Systems. Journal of Computational Physics 227(1), 400–430 (2007)
2. O'Shea, B., Bryan, G., Bordner, J., et al.: Introducing Enzo, an AMR Cosmology Application. Adaptive mesh refinement: theory and applications. Springer Lecture Notes Comput. Sci. Engng., pp. 134–142 (2004)
3. Bryan, G.L., Norman, M.L., Stone, J.M., et al.: A piecewise parabolic method for cosmological hydrodynamics. Comput. Phys. Comm. 89, 149 (1995)
4. Couchman, H.M.P., Thomas, P.A., Pearce, F.R.: Hydra: an Adaptive-Mesh Implementation of P^3M-SPH. Astrophys. J. 452, 797 (1995)
5. Evrard, A.E.: Beyond N-body – 3D cosmological gas dynamics. Monthly Notices Roy. Astronom. Soc. 235, 911 (1988)
6. Springel, V.: The Cosmological Simulation Code GADGET-2. Monthly Notices Roy. Astronom. Soc. 364(4), 1105–1134 (2006)
7. Snytnikov, A.V., Vshivkov, V.A.: A Multigrid Parallel Program for Protoplanetary Disc Simulation. In: Malyshkin, V.E. (ed.) PaCT 2005. LNCS, vol. 3606, pp. 457–467. Springer, Heidelberg (2005)
8. Vshivkov, V.A., Lazareva, G.G., Kireev, S.E., Kulikov, I.M.: Parallel implementation of the gas component model of self-gravitating protoplanetary disc on supercomputers. Vychislitel'nye tehnologii (in Russian) 12(3), 38–52 (2007)
9. Hockney, R.W., Eastwood, J.W.: Computer Simulation Using Particles. IOP Publishing, Bristol (1988)
10. Grigoryev, Yu.N., Vshivkov, V.A., Fedoruk, M.P.: Numerical "Particle-in-Cell" Methods. Theory and Applications. VSP (2002)
11. Belocerkovskiy, O.M., Davydov, Yu.M.: Large particles method in gas dynamics. M.: Nauka (in Russian) (1982)
12. Kireev, S.E.: Parallel implementation of particle-in-cell method for simulation of gravitational cosmodynamics problems. Avtometriya (in Russian) 3, 32–39 (2006)

Supercomputer Simulation of an Astrophysical Object Collapse by the Fluids-in-Cell Method⋆

Igor Kulikov[1], Galina Lazareva[2], Alexey Snytnikov[2], and Vitaly Vshivkov[2]

[1] Novosibirsk State Technical University, Novosibirsk, Russian Federation
[2] Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Novosibirsk, Russian Federation
{kulikov,lazareva,snytav,vsh}@ssd.sscc.ru

Abstract. Parallel implementation of the Fluids-in-Cell Method (FlIC) method is created for 3D cartesian simulation of an astrophysical object collapse. The main parameters of the parallel implementation are given of the FlIC method. The equations under solution are the gas dynamics equations and Poisson equation. Simulation of collapse with FlIC method is compared to SPH simulation. As a result, we can state that FlIC method provided fine enough grid gives better spatial resolution than SPH.

1 Introduction

Collapses of astrophysical objects are under active theoretical study today because of the emergence of a large amount of new observational data. The phenomenon of collapse takes place both at initial and and at the final stages of stellar evolution. An example of the latter are the expansions of supernovae with collapsing core [1]. Simulation in astrophysics is the main method for investigation of nonlinear processes of cosmic structure evolution as well as for verification of theories for the origin of the Universe. At the first place it is necessary to introduce the gas component that interacts with dark matter via gravitation.

At present two methods are mainly in use from the wide variety of the numerical gas dynamics methods. They are the Lagrangian SPH (Smoothed Particle Hydrodynamics) [2] method and Eulerian AMR (Adaptive Mesh Refinement) [3] method. SPH method is based on the interpolation of grid cells in the softening area, and AMR techniques are based on the PPM (piece-parabolic method), which is eesntially a high order Godunov method.

For the hydrodynamical quantities to be defined in SPH method it is necessary to set smoothing length carefully. The smoothing length determines the number of neighbours that affect the SPH particle. One of the features of the method is that the number of neighbours must be kept nearly equal for all the particles in the computational domain in order to gain correct solutions. If the numbers of

⋆ The contents presented in this paper was partially supported by the Research Grants from RFBR 08-01-00615 and Integration Project of SB RAS 103.

neighbours for all the particles differ significantly then the results of computation are not reliable [4]. In order to deal with this problem an adaptive smoothing length is introduced, which allows greater differences in the number of neighbours from one partice to another.

Thus the setting of the smoothing length contains an uncertainty and consequently affects the solution [5]. This is why the increase of the particle number does not necessary lead to the correct solution in the SPH method.

Using AMR one usually sets the value of the finest grid step. It should be noticed that it is difficult to estimate this value because too small grid step might provoke problems if some gravitational instabilities arise. These problems are common for grids with coordinate lines going along the domain boundaries. Even with the finest possible grid steps the shape of the grid cell is still rectangular, and the directions in the space are still not equiprobable, and still some rectangle-shaped structures may appear only due to the grid influence. And in the case of collapse with virtually unlimited density increase the value of the finest grid step cannot be set at all.

The development of eulerian grid methods with no influence of the grid lines on the solution may help to deal with this AMR bottleneck. The development of the grid invariant methods is a more difficult problem than the building of adaptive grids, nevertheless grid invariant methods are possible to create.

Gas dynamics equations are known to be invariant in respect to some point transformation group in the space of dependent and independent variables. This sort of invariance follows from the invariance of the conservation laws, the latter being the basis for gas dynamics equations.

The use of the grid necessarily breaks invariance in the computation [6]. It may be clearly seen, for example, in simulation of the flows that move under different angles in respect to the grid lines. Due to this reason the first differential approximation is traditionally employed for the study of stability and dissipation properties of numerical schemes [7]. Russian scientists [8,9] have adapted the Roe-Osher scheme for the solution of 3D gravitational gas dynamics for ideal gas without AMR methods. The development of the rotation invariant numerical schemes is considered in the following papers [10]. In the paper for construction of finite difference schemes operator approach is proposed that results in symmetrical solutions. The second paper describes a modification of the FlIC method.

The problem is three-dimensional and non-stationary. It implies strict requirements for the method to achieve good results with the limited computer memory and computation time. Rapid progress of computers in recent time enabled to conduct resource-consuming computations and obtain physically valuable results for 3D programs. The use of supercomputers enables to use larger volumes of data and to increase speed and, consequently, precision of computations [11]. It is extremely important for collapse simulation. The main goal of the present work is the study of collapse simulation with the parallel implementations of the grid FlIC method and the gridless SPH method.

2 Numerical Method Description

Let us consider gas dynamics equation system together with Poisson equation in the non-dimensional form:

$$\frac{\partial \rho}{\partial t} + div(\rho \boldsymbol{v}) = 0 \tag{1}$$

$$\frac{\partial \rho \boldsymbol{v}}{\partial t} + div(\boldsymbol{v}\rho \boldsymbol{v}) = -grad(p) - \rho grad(\Phi) \tag{2}$$

$$\frac{\partial p}{\partial t} + div(p\boldsymbol{v}) = -(\gamma - 1)p div(\boldsymbol{v}), \tag{3}$$

$$\Delta \Phi = 4\pi\rho \tag{4}$$

$$p = (\gamma - 1)\rho\epsilon \tag{5}$$

Here ρ is density, $\boldsymbol{v}$ is the velocity vector, p is the pressure, Φ is the gravitational potential, ϵ is the unit density of inner energy, γ is the adiabat index. For transition to the non-dimensional form the following basic quantities were chosen:

– distance from the Sun to the Earth $L = 1.5 \cdot 10^{11} m$
– Solar mass $M_0 = 2 \cdot 10^{30} kg$
– gravitational constant $G = 6.67 \cdot 10^{-11} N \cdot m^2 \cdot kg$

FlIC method was chosen as the basis for solving gas dynamics equations [12]. This method was earlier applied for computations in gas dynamics without gravitation [13]. Thus the method was to be modified for solving graviational gas dynamics problems. The details of modification and numerical implementation are given in [10].

Poisson equation is solved by the Fourier transform method. The 27 point stencil is used for the approximation of Poisson equation. This kind of stencil is required to achieve rotation invariance of the whole method [11].

3 Parallel Implementation

3.1 The Scheme of the Parallel Implementation

FlIC method is a difficult one for an efficient parallel implementation. One of the main questions is the proper distribution of grid arrays between processor elements. In order to create parallel implementation of the FlIC method domain decomposition technique was chosen. FlIC method for gravitational gas dynamics has four stages:

1. Eulerian stage
2. Lagrangian stage
3. Poisson equation solution
4. Gas values recomputation

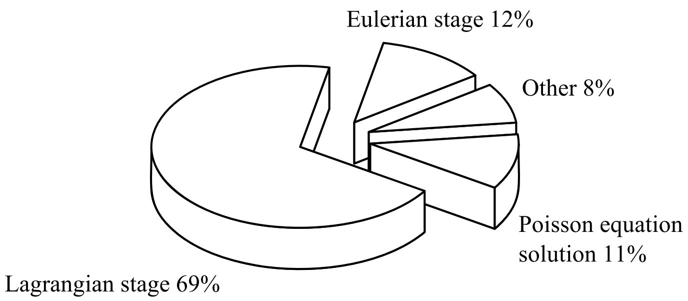


Fig. 1. The portion of each FlIC stage in the total computation time

Relative time (in percent) for each stage during a timestep is shown in figure 1. It should be noted that each stage consists of a constant number of operations and the number of operations does not depend on the solution. Let us consider domain decomposition for each stage. At each stage the computational domain of the size $N_x \times N_y \times N_z$ is divided along OX axis (fig. 2a) into layers with the size $N_x/P \times N_y \times N_z$, here P is the number of processors. Domain decomposition at the Eulerian stage is performed with one-layer overlapping (fig. 2b). One layer is enough because the central difference operator is used for derivative approximation. Lagrangian stage contains the convectional transport of gas values. It is essentially the redistribution of the values between adjacent cells [10]. Since the domain is divided between processors the transport in boundary layers should be computed by both adjacent processors. Thus Lagrangian stage domain decomposition is performed with two-layer overlapping (fig. 2c).

The 3D parallel Fourier Transform is performed with the procedure from the FFTW library [14]. Here the way of dividing the domain into subdomains is set by FFTW itself.

3.2 The Efficiency of the Parallel Implementation

It reqires large amounts of data for the physically meaningful result to be achieved in the gravitational gas dynamics. Thus it is important to estimate what is the maximal size of the problem that allows still to obtain the speedup of the parallel algorithm. The questions is: at what maximal size of the problem we can still decrease computation time by using more processors? In order to solve problems of larger size it will be necessary to find new ways for the parallelization of the method, or even new methods to solve the problem as a whole.

Let us study the efficiency of the parallel implementation with small size problem, that is suitable for the memory of a single processor node of a cluster. Figure 3a gives the dependence of the total computation time on the number of processsors. Figure 3b shows the speedup and figure 3c displays the parallel program efficiency.

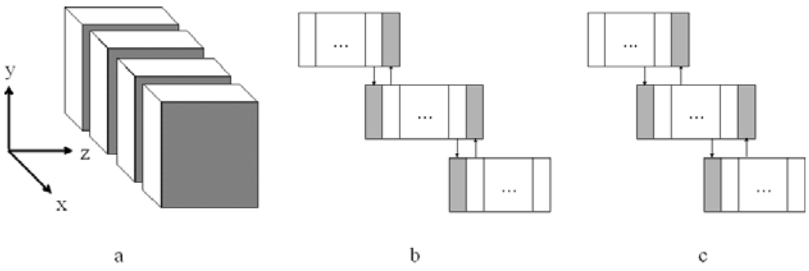


Fig. 2. a) Dividing the computational domain along axis, b) Overlapping of the subdomains at the Eulerian stage, c) Overlapping of the subdomains at the Lagrangian stage

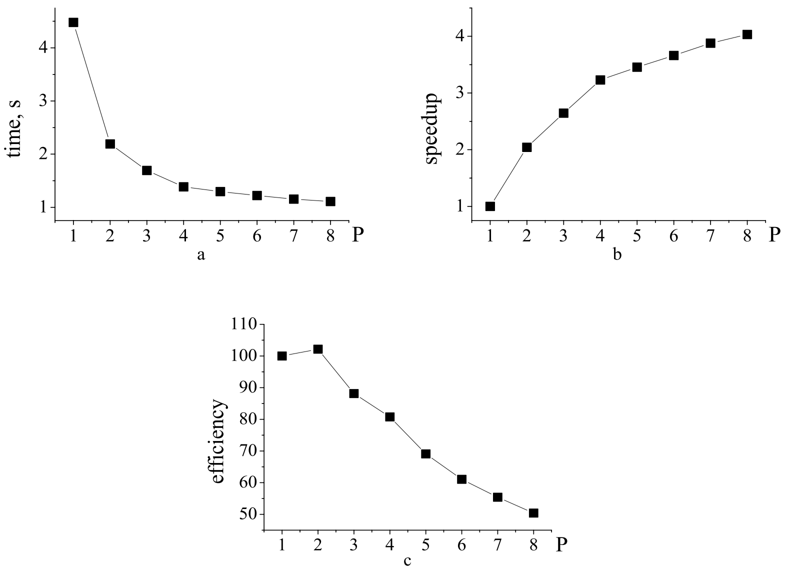


Fig. 3. Computation time (a), speedup (b), efficiency (c) of the parallel program

The given results for speedup and efficiency of the program under study are similar to the properties of other parallel programs that deal with gravitational gas dynamics [11,15,16].

Figure 4 shows the computation time for each of the stages. It is clear from the figure that the speedup of the whole program is mainly defined by the speedup of the Lagrangian stage. Eulerian stage computation as well as gas values recomputation stage reach saturation with 6 or more processors. Saturation means that computation time is no more larger than communication time.

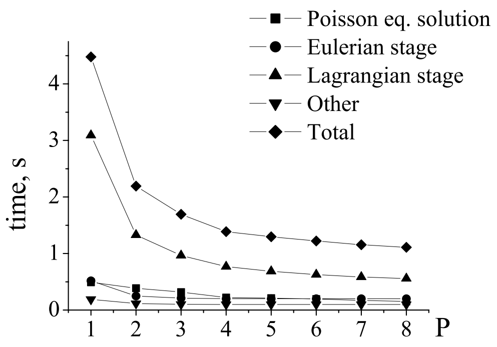


Fig. 4. Computation time for each stage and for method as a whole

4 Computational Experiment

Let us consider the result of collapse simulation. First it is necessary to estimate the precision of the simulation. The main criterion if correctness is the conservation of total energy of the system. Then it will be possible to compare density profiles obtained by the FlIC method (the present implementation) and by SPH method [17]. Initial density profile of the resting gas sphere used for the simulation is shown in fig.5.

Pressure is given by the formula $p = 0.1 \cdot \rho^{\gamma}$, adiabat index is $\gamma = 5/3$.

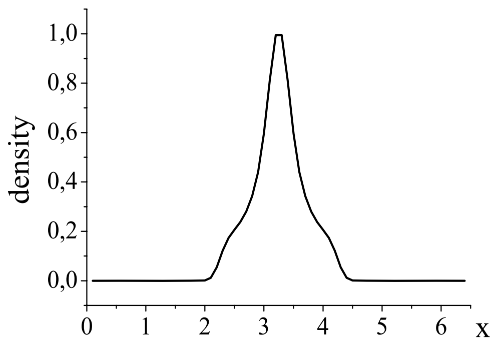


Fig. 5. Initial density distribution

4.1 Precision of Collapse Simulation with FlIC Method

The goal of the present section is the study of the total energy behavior when the number of grid nodes is being increased. The error in the total energy mostly arises at the final stage of collapse when density as well as the other gas values are increased by orders of magintude. In the conducted simulations the main part of the gas mass in the final collapse stage is situated inside the sphere with the radius like $R_{collaps} = 0.1 \cdot R_0$ (10% from the initial radius of the gas sphere). Thus for the correct simulation of the process it is necessary to have enough grid cells at the length of $R_{collaps}$.

Let us compare energy conservation with different number of grid nodes (fig. 6). It is seen from the figure that when the number of nodes is increased the relative error decreases. It means that it is possible to reach the necessary precision provided large enough number of grid nodes. For the grid with $512 \times 512 \times 512$ nodes the error is about 5%. It is tolerant for the comparison of the FlIC solution with SPH solution.

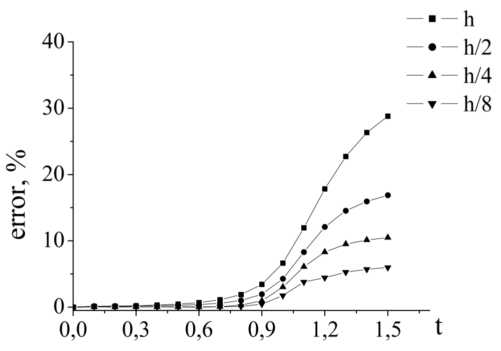


Fig. 6. Error

4.2 Comparison of the FlIC Collapse Simulation with SPH Simulation

Let us consider density profiles in collapse simulation by both FlIC and SPH methods.

It is clear from the comparision of collapse simulation results (fig. 7a and 7b), that the SPH profile is more gently sloping. It is possible that the restriction of SPH method appear in the computation. The bottleneck of SPH method is dealing with high density gradients, since the traditional SPH method provides correct simulation only for small gradients inside the smoothing kernel. And if the smoothing length for SPH particle is made very small to make the local gradients low, then there would be problems to provide nearly equal number of neighbours for all particles.

As it is seen from the conducted computations in the case of collapse simulation the gradients can not be made low. Thus it is hard for traditional SPH method to provide correct simulation. FlIC method simulates collapse with the error of 5% in total energy provided 10 grid nodes for the length $R_{collaps}$ in the final collapse stage. Let us note that mass distributions (fig. 7c) are in good agreement with each other.

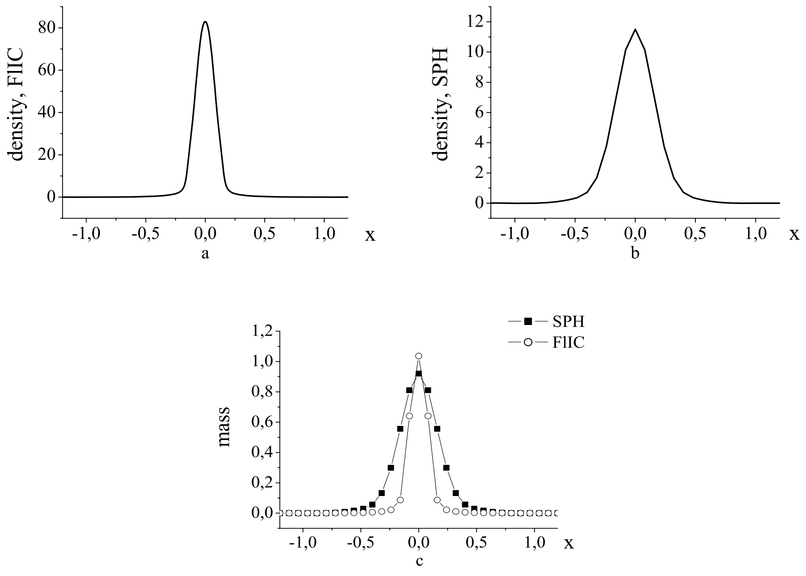


Fig. 7. Density profiles obtained by FlIC method (a), SPH method (b), and mass distribution (c)

5 Conclusion

The parallel implementation of FlIC method described in the paper is capable of obtaing correct results for 3D gravitational gas dynamics. One of the main problems is the distribution of grid arrays between procesors with the necessary ovelapping. Test computations were conducted with the supercomputers NKS-160 (Siberian Supercomputer Centre, Novosibirsk, Russia) and MVS-6000 (Joint Supercomputer Centre, Moscow, Russia). These computations did not show linear speedup for gas dynamics equation solution because of the saturation due to the high communication time.

The parallel implementation of gravitational gas dynamics enabled to use finer grids for computation and to solve a wider variety of problems. As an example, the steep gradients are hard to deal in SPH, but it is shown for the simulation of collapse with FlIC method that setting large enough number of grid nodes will provide the necessary precision.

References

1. Ardeljan, N.V., Bisnovatyi-Kogan, G.S., Kosmachevskii, K.V., Moiseenko, S.G.: An implicit Lagrangian code for the treatment of nonstationary problems in rotating astrophysical bodies. Astron. Astrophys. Suppl. Ser. 115, 573–594 (1996)
2. Monaghan, J.J., Gingold, R.A.: Shock simulation by the particle method SPH. J. Comp. Phys. 52, 374–389 (1983)

3. Collela, P., Woodward, P.R.: The piecewise parabolic method (PPM) for gas-dynamical simulations. J. Comp. Phys. 54, 174–201 (1984)
4. Attwood, R.E., Goodwin, S.P., Whitworth, A.P.: Adaptive Smoothing Length in SPH. Astron. Astrophys. 464, 447–450 (2007)
5. Hubber, D.A., Goodwin, S.P., Whitworth, A.P.: Resolution requirements for simulating gravitational fragmentation using SPH. Astron. Astrophys. 450, 881–886 (2006)
6. Paasonen, V.I., Shokin, Yu.I., Yanenko, N.N.: On the theory of difference schemes for gas dynamics. Lect. Not. Phys. 35, 293–303 (1975)
7. Shokin, J.: On the First Differential Approximation Method in the Theory of Difference Schemes for Hiperbolic Systems of Equations. Amer. Math. Society (1973)
8. Kaigorodov, P.V., Kuznetsov, O.A.: Adaptation of Roe-Osher Scheme for the Computers with Massive-Parallel Architecture. KIAM Preprint 59 (2002)
9. Bisikalo, D.V., Boyarchuk, A.A., Kaygorodov, P.V., Kuznetsov, O.A., Matsuda, T.: The Structure of Cool Accretion Disc in Semidetached Binaries. Astron. Rep. 81, 494–502 (2004)
10. Vshivkov, V.A., Lazareva, G.G., Kulikov, I.M.: A modified fluids-in-cell method for problems of gravitational gas dynamics. Optoelectronics, Instrumentation and Data Processing 43, 530–537 (2007)
11. Kireev, S., Kuksheva, E., Snytnikov, A., Snytnikov, N., Vshivkov, V.: Strategies for Development of a Parallel Program for Protoplanetary Disc Simulation. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 128–139. Springer, Heidelberg (2007)
12. Grigoryev, Y.N., Vshivkov, V.A., Fedoruk, M.P.: Numerical "Particle-in-Cell" Methods. Theory and applications, Utrecht-Boston (2002)
13. Flow Vision Home Page, http://www.flowvision.ru
14. FFTW Home Page, http://www.fftw.org
15. Snytnikov, N., Vshivkov, V., Snytnikov, V.: Study of 3D Dynamics of Gravitating Systems Using Supercomputers: Methods and Applications. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 162–173. Springer, Heidelberg (2007)
16. Snytnikov, A., Vshivkov, V.: A multigrid parallel program for protoplanetary disc simulation. In: Malyshkin, V.E. (ed.) PaCT 2005. LNCS, vol. 3606, pp. 457–467. Springer, Heidelberg (2005)
17. Springel, V., Yoshida, N., White, S.: GADGET: A code for collisionless and gas-dynamical cosmological simulations. New Astronom 6, 79–117 (2001)

High-Performance Tsunami Wave Propagation Modeling

Mikhail Lavrentiev-jr[1,2], Alexey Romanenko[2], Vasily Titov[3], and Alexander Vazhenin[4]

[1] Sobolev Institute of Mathematics of Russian Academy of Science, Novosibirsk, Russia
mlavr@nsu.ru
[2] Novosibirsk State University, Novosibirsk, Russia
arom@ccfit.nsu.ru
[3] NOAA/Pacific Marine Environmental Laboratory, Seattle, WA, USA
vasily.titov@noaa.gov
[4] University of Aizu, Aizu-Wakamatsu, Fukushima, Japan
vazhenin@u-aizu.ac.jp

Abstract. Strongest earthquake of December 26, 2004 generated catastrophic tsunami in Indian Ocean. This shows that, in spite of recent technology progress, population at coastal zone is not protected against tsunami hazard. Here, we address the problem of tsunami risks mitigation. Note that prediction of tsunami wave parameters at certain locations should be made as early as possible to provide enough time for evacuation. Modern computational technologies can accurately calculate tsunami wave propagation over the deep ocean provided that initial displacement (perturbation of the sea bed at tsunami source) is known. Modern deep ocean tsunameters provide direct measurement of the passing tsunami wave in real time, which help to estimate initial displacement parameters right after the tsunami wave is recorded at one of the deep ocean buoys. Therefore, fast tsunami propagation code that can calculate tsunami evolution from estimated model source becomes critical for timely evacuation decision for many coastal communities in case of a strong tsunami. Numerical simulation of tsunami wave is very important task for risk evaluation, assessment and mitigation. Here we discuss a part of MOST (Method of Splitting Tsunami) software package, which has been accepted by the USA National Ocean and Atmosphere Administration as the basic tool to calculate tsunami wave propagation and evaluation of inundation parameters. Our main objectives are speed up the sequential program, and adaptation of this program for shared memory systems (OpenMP) and CELL architecture. Optimization of the existing parallel and sequential code for the task of tsunami wave propagation modeling as well as an adaptation of this code for systems based on CELL BE processors (e.g. SONY PlayStation3) is discussed. The paper also covers approaches and techniques for programs optimization and adaptations, and obtained results.

Keywords: Tsunami Wave Propagation Modeling, Method of Splitting Tsunami, Fine-grain Parallel Algorithms, OpenMP Paradigm, Cell BE Architecture.

1 Introduction

Shallow water approximations (both linear and nonlinear) are considered worldwide as accurate propagation models for tsunami waves. These models describe reasonably well waves parameters (both traveling times between all recorded sources and available measurement stations and amplitudes) even for rather rough digital bathymetry, provided that the initial sea bed displacement at source is given. Several software packages have been proposed to simulate wave propagation over the ocean and inundation zones. Accordingly, we can distinguish several approaches related to the practical realization of those packages.

The method described in [1] is oriented to create a parallel hybrid tsunami simulator that can mix different models, methods and meshes, maybe even incorporate "alien software". This goal is achieved by combining overlapping domain decomposition and object-oriented programming. Actually, the computing performance is not a main goal of this approach.

In paper [2], eight different parallel implementations were used of the shallow water equations to simulate the tsunami model. Each of these implementations used a mixed-mode programming model from thread based shared memory, to distributed memory and finally to a virtual shared memory. As shown in this paper, scalability issues become paramount, and threading becomes a significant bottleneck if sufficient node memory is not available.

TUNAMI-N2 [3] is a tsunami numerical simulation program with the linear theory in deep sea and with the shallow water theory in shallow sea and on land with constant grid size in the whole region. TUNAMI was originally authored by Imamura in 1993 for the Tsunami Inundation Modeling Exchange (TIME) program, and has been applied to several tsunami events.

MOST (Method of Splitting Tsunami) [3-4], developed at Pacific Marine Environmental Laboratory (NOAA, Seattle, USA), allows for real time tsunami inundation forecasting by incorporating real-time data from detection buoys. The model MOST is also used in the United States for developing inundation maps as well as for Tsunami Inundation Modeling [6]. The new web enabled interface for MOST is released with the name ComMIT. That is why that an acceleration of executable code of the MOST package should provide additional time for tsunami hazard mitigation.

Here we present results of investigations of a performance gain, obtained with the help of parallel technologies and devoted to fine-grained parallelization of a part of the MOST software that is used for calculating the tsunami wave propagation over the deep ocean. Particularly, the presented paper is focused on design and comparative analysis of parallel algorithms for the OpenMP platform and the most attractive today IBM Cell BE architecture.

The rest of this is organized as follows. Section 2 explains the mathematical model for simulating wave propagation used in the MOST software including analysis of numerical data needed for numerical modeling. In Sections 3, we describe sequential and Open MP algorithms. Thereafter, Section 4 presents some preliminary parallel simulation results for the Cell BE algorithms. Finally, some concluding remarks and comments about future work are given in Section 5.

2 Theoretical Backgrounds and Data Sets Analysis

First algorithms of MOST application have been designed in the Siberian Division of the Russian Academy of Sciences (Novosibirsk, Russia) late in 80-s by Titov. Currently it is among the advanced tools for tsunami simulation. The MOST environment includes numerical simulation codes capable of simulating all processes of tsunami evolution: earthquake, transoceanic propagation, and inundation of dry land. Here, the part is only discussed of the MOST software for calculating the wave propagation in a deep ocean.

2.1 Mathematical Model

Wave propagation is simulated by nonlinear hyperbolic shallow water system

$$\begin{aligned} &H_t + (uH)_x + (vH)_y = 0, \\ &u_t + uu_x + vu_y + gH_x = gD_x, \\ &v_t + uv_x + vv_y + gH_y = gD_y, \end{aligned}$$

where H(x, y, t) = η(x, y, t) + D(x, y, t) - stands for wave height, calculated from the undisturbed sea level; D – depth profile (digital bathymetry); u(x, y, t), v(x, y, t) – velocity components along x and y axis, respectively; g – acceleration of the gravity.

The above system could be presented in matrix form as

$$z = \begin{pmatrix} u \\ v \\ H \end{pmatrix}, \quad A = \begin{pmatrix} u & 0 & g \\ 0 & u & 0 \\ H & 0 & u \end{pmatrix}, \quad B = \begin{pmatrix} v & 0 & 0 \\ 0 & u & g \\ 0 & H & u \end{pmatrix}, \quad F = \begin{pmatrix} gD_x \\ gD_y \\ 0 \end{pmatrix} \tag{1}$$

$$\frac{\partial z}{\partial t} + A\frac{\partial z}{\partial x} + B\frac{\partial z}{\partial y} = F$$

Splitting method for numerical treatment is based on two auxiliary systems, each depending on one spatial variable:

$$\frac{\partial \varphi}{\partial tt} + A\frac{\partial \varphi}{\partial tx} = F_1, \quad 0 \le x \le X \qquad \frac{\partial \psi}{\partial tt} + B\frac{\partial \psi}{\partial ty} = F_2, \quad 0 \le y \le Y$$

$$F_1 = \begin{pmatrix} gD_x \\ 0 \\ 0 \end{pmatrix}, \quad F_2 = \begin{pmatrix} 0 \\ gD_y \\ 0 \end{pmatrix}.$$

Each system could be transformed into canonical form:

$$\begin{aligned} &v'_t + \lambda_1 v'_x = 0, \\ &p_t + \lambda_2 p_x = gD_x, \\ &q_t + \lambda_3 q_x = gD_x, \end{aligned} \tag{2}$$

where

$$v' = v,$$
$$p = u + 2\sqrt{gH},$$
$$q = u - 2\sqrt{gH}.$$

are invariants of the system (1) and

$$\lambda_1 = u, \quad \lambda_{2,3} = u \pm \sqrt{gH}.$$

Numerical algorithm for solution of the shallow water equations system (1) is described as follows: After input data and variables initialization the girded variables are to be determined by the finite difference scheme. Each time step of this algorithm is splitting by two sub-steps. Each sub-step consists of determination of invariants *v', p, q* using original variables values and of sequential solution of canonical system along X and Y axis directions. After completion of the time-step the values of original variables *u, v, H* must be restored using new values of invariants.

The characteristic line method has been used to set the boundary conditions for the system. At the open sea boundary the conditions

$$v' = 0, r = \pm 2\sqrt{gD}\text{ , where } \boldsymbol{r} = \boldsymbol{u} \pm 2\sqrt{\boldsymbol{gH}} \text{ (i.e. } r = p \text{ or } q\text{)}$$

are used. At the land boundary the conditions of perfect reflection are used:

$$v = 0, \ p = -q.$$

For the numerical solution of the system the following finite difference scheme is used:

$$\frac{\vec{W}_i^{n+1} - \vec{W}_i^n}{\Delta t} + A\frac{\vec{W}_{i+1}^n - \vec{W}_{i-1}^n}{2\Delta x} - A\Delta t\frac{A(\vec{W}_{i+1}^n - \vec{W}_i^n) - A(\vec{W}_i^n - \vec{W}_{i-1}^n)}{2\Delta x^2} =$$
$$= \frac{\vec{F}_{i+1} - \vec{F}_{i-1}}{2\Delta x} - A\Delta t\frac{\vec{F}_{i+1} - 2\vec{F}_i + \vec{F}_{i-1}}{2\Delta x^2},$$

where

$$A = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}, \quad \vec{W} = (v', p, q), \quad \vec{F} = (0, gD_x, gD_x)$$

It could be observed that calculations along coordinates could be performed independently. This suggests performance gain through program parallelization.

2.2 Data Sets and Values

Let us estimate the data volume needed for successful modeling. For the typical digital bathymetry it is enough to have the distance between the mesh nodes about 3,6 km. Therefore, the computational domain should have a size of 2048x2048 points to cover the Pacific Ocean. Accordingly, the NOAA uses 2500x1800 mesh size. Taking into account this mesh size, it is possible to estimate the time complexity of the MOST method. The complete wave propagation modeling requires implementing calculations for multiple time steps. The required number of such time steps is about 1440 to cover for 24 hours time period.

Importantly, our investigations were oriented to accelerate calculations of the single time step. Therefore, this problem can be considered as a fine-grain task. That is why the use of the shared memory programming paradigm seems to be very are attractive in solving this problem. The total amount of data is no more that 64Mb. The operation complexity is about 10^9 operations fro each time step.

3 Sequential and OpenMP Algorithms

3.1 Sequential Programs

Taking into account the mentioned above possibility to provide calculation independently along the X and Y coordinates, the following computing strategy could be used.

```
Calculations (for each time step)
1. Do calculation along X axis
  1. Prepare invariants along X axis for each row
  2. Do calculations (use invariants only) - swater
function
  3. Convert new values of invariants back to physical
quantities
1. Do calculation along Y axis
  1. Prepare invariants along Y axis for each row
  2. Do calculations (use invariants only) - swater
function
  3. Convert new values of invariants back to physical
quantities
```

Fig. 1. The main calculation loop

The original code of the MOST software was implemented on Fortran 90. It takes 3.31 seconds for one time step on 4 dual-core CPU server, based on Intel Xeon, 2.8GHz. After porting this program onto a C/C++ language, it takes about 3.00 seconds to process the one time step. This time well use as a basis for comparison with the parallel OpenMP and IBM Cell BE algorithms. To be able to have a portable version that is suitable for the client-server programming model, this program was also adopted to a Java environment with the calculation time about 18.5 seconds for one time step. The Java class diagram is presented in Fig. 2. This allows also embedding this program into a GUI environment as well as combining it with the visualization tools.

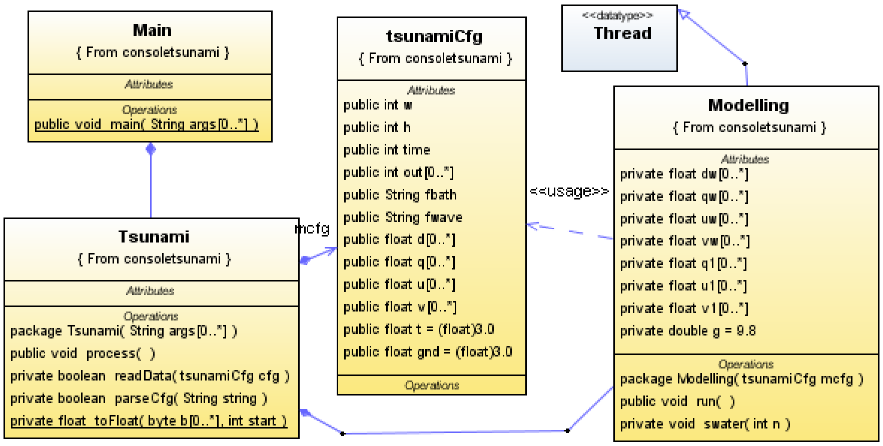


Fig. 2. Java Class Diagram

3.2 Implementation on the OpenMP Platform

The OpenMP paradigm [7] is mainly focused on performing loop iterations in parallel. Actually, a programmer should only point loops in a sequential program that can be processed in parallel. Fig. 3 depicts results of speedup obtained by following to this simple strategy (graph "OpenMP") that was used for loops providing calculations along X and Y axes (Fig. 1). The unstable and relatively small speedup is obtained because of different directions of scanning matrix structures along X (row-wise scanning) and Y (column-wise scanning) coordinates. This leads to an ineffective usage of a cash memory in CPUs.

To keep such a scanning regularity for both computational steps, the intermediate transposing was introduced of matrix structures. Actually, it is necessary to implement forward and back transpositions of four matrices. Importantly, matrix transpositions were implemented in parallel using OpenMP operations based on a block-wise matrix transposition. To optimize the block size, several experiments have been carried out. This allowed achieving the maximum of performance with block size of 64x64 elements. Results presented in Fig. 3 (Graph "+Transposition") show that this strategy allowed to obtain stable speedup growing even that additional time have been spend to implement matrix transpositions.

The last step of the performance optimization is in implementing a part of executable code using on embedded SIMD-stream co-processors. From the programmer's point of view, they can be programmed using special built-in CPU stream operations known as SSE instructions. This part of executable code has been rewritten using SSE instructions for calculations of invariants, height of wave and its speed along axis. Results the final parallel program implementation are shown in Fig. 3 (graph "+SSE"). The best result is about ten times speedup for 5 cores in comparison with the sequential program (Fig. 1). Performance decrease with more then 5 cores is

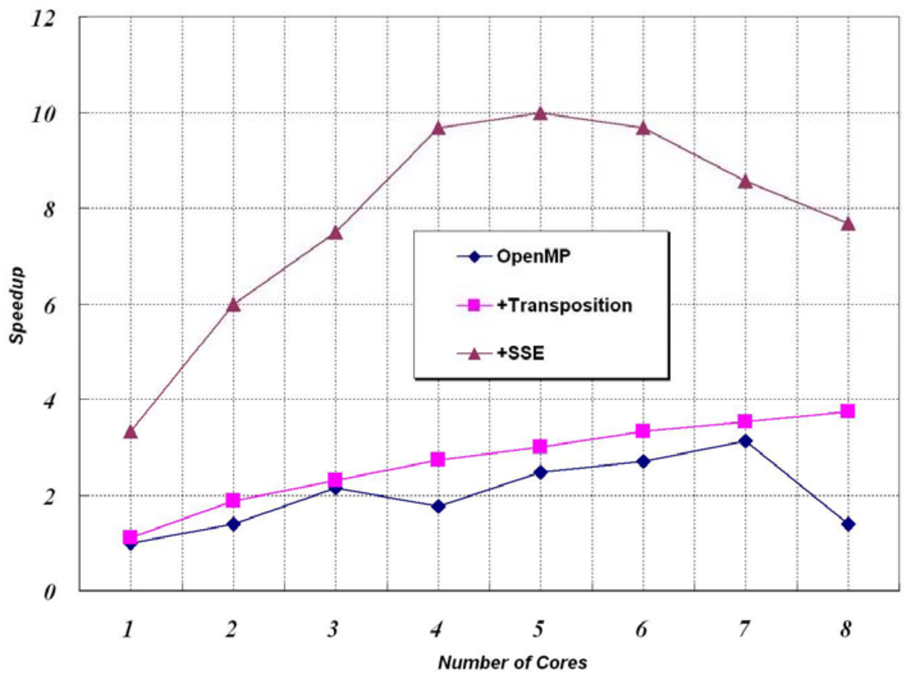


Fig. 3. Speedup of Several OpenMP Optimizations

because of using a smaller amount of data to be processed, and a prevalence of communications over the computations in CPUs. As was pointed in Subsection 2.2, we use the fixed mesh size required by acceptable result accuracy. Actually, here we have additional possibilities to increase performance by increasing a size of mesh. To analyze the accuracy of parallel algorithms, a set of numerical experiments are also provided with different types of data distributions among the processes. The final result fluctuates in the acceptable range of error measurements.

4 Algorithms for the IBM Cell BE Architecture

The Cell Architecture combines the considerable floating point resources required for demanding numerical algorithms with a power efficient software-controlled memory hierarchy [8,9]. Despite its radical departure from previous mainstream/commodity processor designs, the Cell is particularly compelling because it will be produced at such high volumes that it will be cost competitive with commodity CPUs. Example of this approach can be the SONY PlayStation 3, the architecture of which includes the Cell Broadband Engine (Cell BE) Processor. Cell BE processor is targeting for a wide range of electronic devices (from portable/handheld PDAs to supercomputers), offering high performance for computer entertainment, virtual-reality, wireless communication, real-time video, interactive TV, and other high-performance applications. Cell has a reputation of being difficult to program for. This is true in the sense that it is 'more involved' or 'requires a different way of thinking'. The IBM provides users with CELL SDK so users could write there own applications for this processor.

From the programmer's point of view, the Cell BE architecture BE can be presented as shown in Fig. 4. The Cell computational resources can be divided into two parts. The Power Processor Element (PPE) is a conventional processor the main responsibility of which to set up tasks for the SPE cores. In a Cell based operating system (OS), the PPE runs the OS kernel, service program and the most of the user's applications. Each Synergistic Processor Element (SPE) is a RISC processor with 128-bit SIMD organization for single and double precision instructions. SPEs compute intensive parts of the OS and applications. The SONY PlayStation3 allows the user to use six SPE only. Two SPEs are reserved for OS needs. Taking into account the OpenMP algorithms described above, the design of effective Cell algorithms was mainly focused on distributing iterations along axis among SPEs.

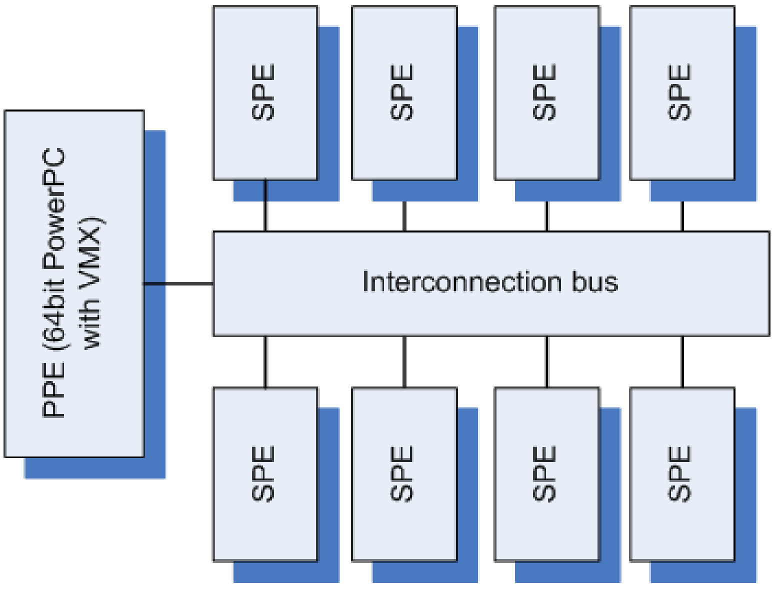


Fig. 4. CELL BE CPU architecture

Actually, it is necessary to pass over several steps in order to develop effective Cell algorithms. The first step is in porting a sequential program and running it on PPE. An average PPE time for the one time step is about 7.5 seconds. We will use this time as a basic value. The next step is in reorganizing a program in order to assign tasks for SPEs. Fig. 5 shows results of comparison between two strategies. The first strategy is when the PPE acts as a master sending tasks to SPEs. The second one is when SPEs take their sub-tasks by themselves. As shown in Fig. 5, the PPE hardly could have enough time to manage all SPEs, because processing time is rather small. When processing time significantly exceeded a time for data preparation this data distribution strategy can be more effective. As it is mentioned in [9], it is necessary to combine

I/O and processing operations for increasing the SPE performance. This can be achieved by using two buffers of data in SPE: one buffer is for data processing, and the other one is to prepare next portion of the data. The small performance increase for the dual buffer implementation is that the data transfer rate is disparagingly taking a little time in comparison with the processing time.

Fig. 5 shows that the speedup value is still on the very low level even that its growth is rather stable. That is why the last step of the code optimization is devoted to realize the most intensive computational parts by means of the SIMD and pipelining strategies inside each SPE. Time measurements showed that the function call **swater** takes 97% of the program execution time. The other investigations showed that while the number of involved SPE is more than two, even pipeline (it response for calculations) idles for 98% of the time because of the access to the local memory and data conversion by odd pipeline. Optimization of the **swater** function implementation using the vectorization of calculations allowed to obtain the calculation speedup shown in Fig. 6.

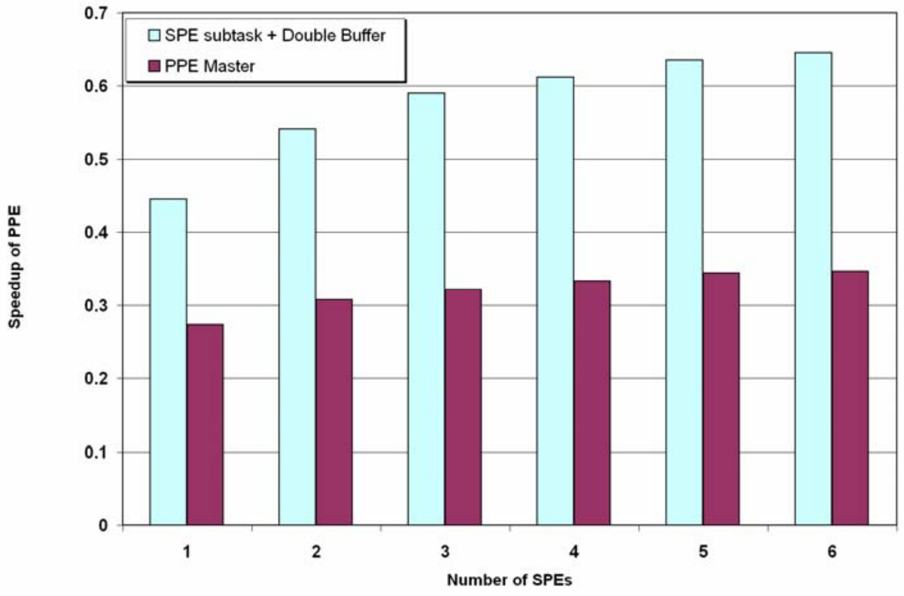


Fig. 5. Parallel Cell Processing without Optimization

This amazing jump in performance was achieved by the following reasons:

- Code vectorization decreases the number of operations by the factor 4. Operations over four floats are executed in parallel, which increase the number of elementary operations from one to four per computer clock. This is look like applying SSE instructions in general-purpose CPUs.
- SPEs only support quad-word loads and stores. Scalars lead to significant downtime of pipeline.
- More efficient use of the registers takes place.

The final block diagrams for SPE and PPE are presented in Fig. 7 and 8.

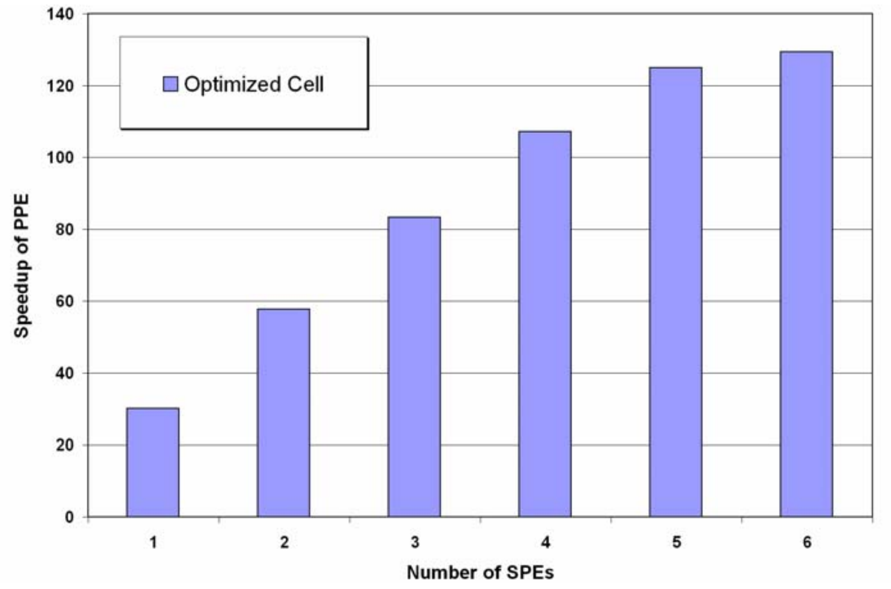


Fig. 6. Speedup of the Optimized Cell Algorithm

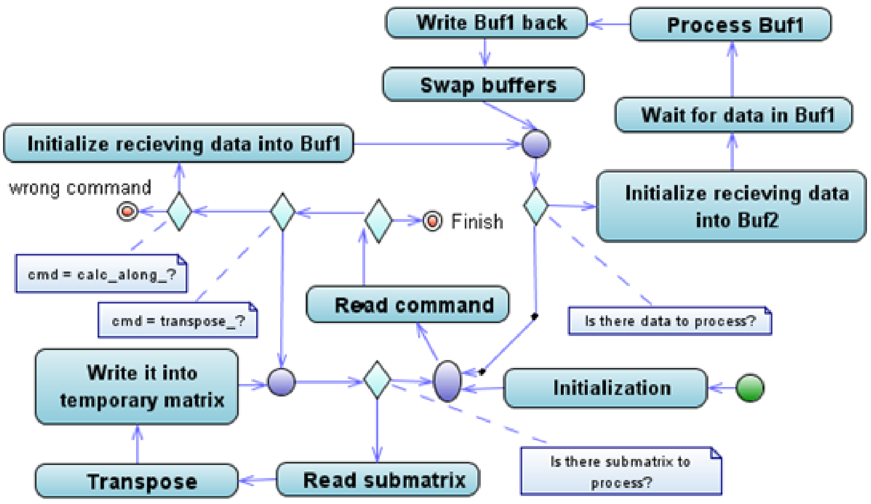


Fig. 7. Block diagram for SPE

Taking into account the features of computer hardware used, the best results achieved for a time needed to calculate the one time step are as follows:

- 0,9 sec for MPI (8 node Linux cluster, Dual Intel PIII-933MHz, RAM - 1GB network – FastEthernet),
- 0,31 sec for the OpenMP (5 cores, 4 dual-core Intel Xeon, 1.86GHz),
- 0,06 sec for the CELL BE (6 SPEs).

Taking into account a problem complexity (about 10^9 operations), the maximum performance is about 17 GFLOPS for the single time step using 6 SPE (Fig. 9). It is more than 120 times more powerful than the original version for CELL PPE processor, more than 5 times faster than a parallel version optimized for 4 cores for OpenMP as well as more than 50 times faster than the original sequential version.

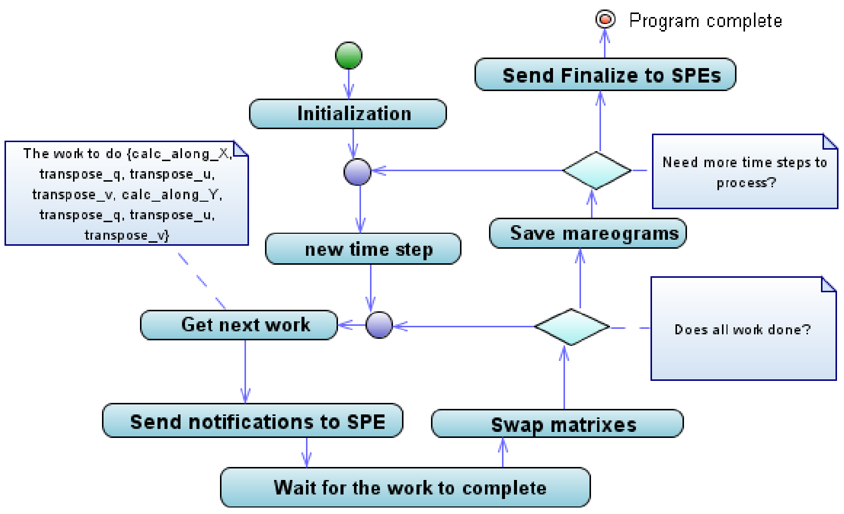

Program complete
Send Finalize to SPEs
Initialization
The work to do {calc_along_X, transpose_q, transpose_u, transpose_v, calc_along_Y, transpose_q, transpose_u, transpose_v}
Need more time steps to process?
Save mareograms
new time step
Get next work
Does all work done?
Send notifications to SPE
Swap matrixes
Wait for the work to complete

Fig. 8. Block diagram for PPE

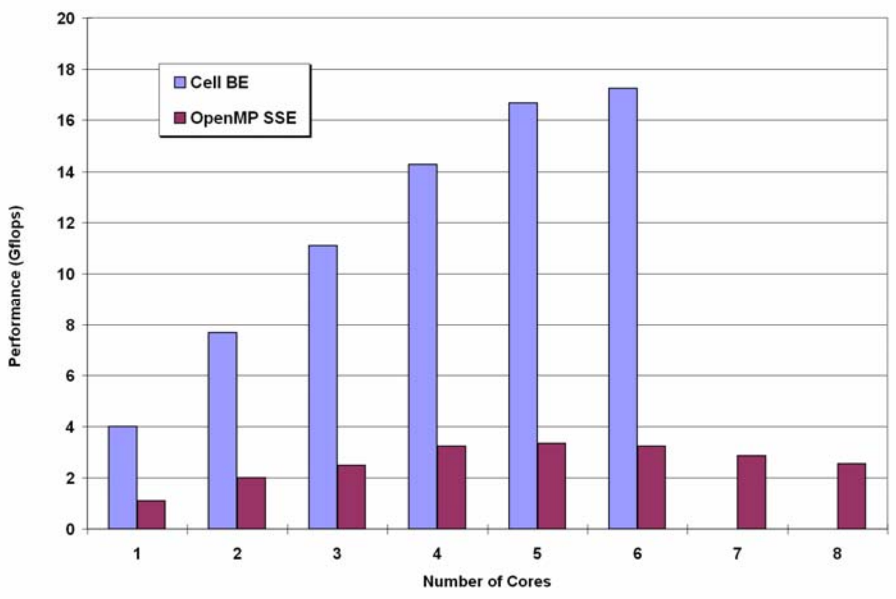


Fig. 9. Performance Evaluation for OpenMP and Cell BE Platforms

5 Conclusion

The Tsunami Wave Propagation module of the MOST was adapted for different parallel environments including Shared memory systems using OpenMP, Distributed memory systems using MPI, and the IBM CELL BE. This kind of problem can be characterized as rather difficult for parallel fine-grained implementation because of necessity to use intermediate matrix transpositions in order to keep data streams on a regular level. Nevertheless, we showed the possibility of a significant acceleration of this problem solution time for different programming platforms including a modern Cell environment with non-standard high-performance equipment like Sony PlayStation3. That is why we consider this program as a portable and acceptable for organizing so-called "public" client-server computing on which the users can grant a free computer time for calculations or obtain results of possible tsunami behaviors. The key point of these future investigations can be a concept of "multi method" technique for better determination of the initial displacement parameters by the recorded time series processing combining with real time signal processing and technique of "calculations in advance".

References

1. Cai, X., Langtangen, H.P.: Making Hybrid Tsunami Simulators in a Parallel Software Framework. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 686–693. Springer, Heidelberg (2007)
2. Ganeshamoorthy, K., Ranasinghe, D.N., Silva, K.P.M.K., Wait, R.: Performance of Shallow Water Equations Model on the Computational Grid with Overlay Memory Architectures. In: The Second International Conference on Industrial and Information Systems (ICIIS 2007), pp. 415–420. IEEE Press, Sri Lanka (2007)
3. Shuto, N., Imamura, F., Yalciner, A.C., Ozyurt, G.: TUNAMI N2: Tsunami Modeling Manual, `http://tunamin2.ce.metu.edu.tr/`
4. Titov, V.V.: Numerical Modeling of Tsunami Propagation by using Variable Grid. In: The IUGG/IOC International Tsunami Symposium, pp. 46–51. Computing Center Siberian Division USSR Academy of Sciences, Novosibirsk, USSR (1989)
5. Titov, V., Gonzalez, F.: Implementation and Testing of the Method of Splitting Tsunami (MOST). Technical Memorandum ERL PMEL-112, National Oceanic and Atmospheric Administration, Washington DC (1997)
6. Borrero, J.C., Sieh, K., Chlieh, M., Synolakis, C.E.: Tsunami Inundation Modeling for Western Sumatra. Procedings of the National Academy of Sciences of the USA 103 (52) (2006), `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1750885`
7. Chandra, R.: Parallel Programming in OpenMP. Morgan Kaupmann Publishers, San Francisco (2001)
8. IBM Research - Cell. IBM, `http://www.research.ibm.com/cell/`
9. Cell Broadband Engine, Programming Handbook. IBM (2007)

Parallel Object Motion Prediction in a Robotic Navigational Environment

Vijay S. Rajpurohit[1] and Manohara Pai M.M.[2]

[1] Department of Computer Science and Engg.
[2] Department of Information and Communication Technology,
Manipal Institute of Technology, Manipal 576 104 India
vijaysr2k@yahoo.com, mmm.pai@manipal.edu

Abstract. In a dynamic Robot navigation system , the Robot has to deal with multiple number of moving objects in the environment simultaneously. The control loop of Robot motion planning comprising of sense-plan-act cycle has very short duration . Predicting the next instance position (Short Term Prediction) and the trajectory (Long Term Prediction) of moving objects in a dynamic navigation system is a part of sense-plan-act cycle. With increase in the number of moving objects under observation, the performance of the prediction techniques reduce gradually. To overcome this drawback, in this paper we propose a parallel motion prediction algorithm to keep track of multiple number of moving objects within the Robotic navigational environment. The implementation of parallel algorithm is done on a cluster computing setup. Performance of the algorithm is tested for different test case scenarios with detailed analysis on efficiency and speedup.

Keywords: Dynamic motion prediction, Short term prediction, Long Term Motion Prediction, Fuzzy rule base, Cluster Computing, Parallel motion prediction.

1 Introduction

In a mobile Robot navigation system predicting the next instance position (Short Term) of moving objects and identifying the trajectory of their motion (Long Term) is one of the essential requirements for obstacle avoidance and safe path planning towards destination. The success of the motion prediction techniques largely depend on the response time of the prediction techniques applied and the temporal validity of the predictions done. In other words one needs to plan motion prediction fast but not very far in the future.

Research literature has addressed various solutions to the short term[1][4][7] as well as long term motion predictions [2][3][9]. Usual tendency of the prediction algorithms proposed in the literature is to consider the nearest object among all the objects under observation for future prediction. This may lead to situations like i)searching for the nearest object among all the objects under observation, which takes considerable amount of time in prediction process. ii) When multiple objects are at the same distance but at different locations, difficulty in selection of the most critical object iii)Poor decisions regarding the future move of the Robot.

Several researchers have proposed to exploit parallel computing techniques to solve Robot motion planning problems[5][6][10]. In our previous work two kinds of object motion predictions have been considered for short term and long term predictions. A Fuzzy based predictor for short term motion prediction algorithm[11] and Fuzzy Self Organizing Map based long term motion prediction algorithm [12]. In this paper, our previous work is extended to allow the Robot to handle multiple number of objects simultaneously using parallel implementation. In the proposed approach both algorithms are integrated into the parallel motion prediction model. Both algorithms in their sequential forms are tested for real life data sets and have shown improved response time and better quality of prediction over the other prediction techniques in various situations. The parallel implementation of the proposed algorithm is done on cluster computing setup using MPI instructions.

The paper is organized as follows. In Section 2 an overview of the Short term and Long term motion prediction algorithms included in the proposed work is given. Parallel computing architecture and algorithm for the proposed work is presented in Section 3. Experimental results are elaborated in section 4. Conclusions are given in Section 5.

2 Moving Object Motion Prediction

It is desirable for the Robot to perform obstacle avoidance in a manner that resembles the human motion for obstacle avoidance. The Robot should act before the obstacles come too close. For this reason, future motion prediction of obstacles is employed.

Short Term motion prediction in the proposed parallel prediction algorithm is implemented using Fuzzy rule based motion prediction algorithm [11]. Positions of moving object in the navigational environment are sampled at equal time intervals and form the input to the Fuzzy Rule base. Next instance object position is predicted using Fuzzy inference process with Mean of Maxima defuzzification. The parallel prediction algorithm incorporates the Fuzzy based Self Organizing Map(FSOM) algorithm[12] for long term motion prediction. The navigational environment under observation is Fuzzy based, representing the object's position (Range and Direction) from the Robot in the form of Fuzzy values. Similar motion patterns observed in the environment are clustered in the learning stage using Self Organizing Map(SOM) and during estimation stage the SOM classifies the partially observed object's trajectory in the environment to one of the clusters generated in the learning stage. The mean trajectory of the cluster identified is the possible future trajectory of the moving object.

3 Parallel Motion Prediction

In a real life scenario the Robot may encounter multiple number of objects at the same time. To process all the objects in the scene simultaneously, the Robot has to process all the objects data in sequential fashion,and should generate results in quick succession such that the results are valid in real time. As the number of objects in the scene increase the response time of the predictor increases for the objects processed at the end. When the Robot encounters n number of objects in the navigation environment, the expected response time of the n^{th} object is given by

$$ShortTerm + LongTerm = \delta_t \; + \delta_{t1} \; + \delta_{t2} \; + \delta_{t3}.........\delta_{tn} \tag{1}$$

Where δ_t δ_{t1}, δ_{t2} .....δ_{tn} represent the time gap between first two sensor readings, the time needed for predicting the next instance position and the trajectory of object1, object2 .... Object_n respectively.

Each object's data is independent of other objects data in the navigational environment and creates a scope for parallel processing. The parallel algorithm is designed to work with a cluster of computers connected in parallel. Each node in the cluster processes a single objectś data to predict its its next instance position and the trajectory. The parallelization of the algorithm is done using Message Passing Interface (MPI).The algorithm initializes the parallel programming state. The Server receives the sensor data of all the objects observed by the vision sensor. Each slave processor(c2 to cn) has the copy of prediction algorithm comprising of both short term and long term prediction algorithms. The Server sends each objectś data to a unique processor in the cluster and one of the objectś data for itself. Each processor executes predictor algorithm with received objectś data as input and calculates the next instance position and the future trajectory of that object. Results of all the processors are are sent back to the Server. Figure 1 represents the cluster of computers (c1 to cn) with each node of the cluster having a separate copy of the predictor program.

3.1 The Parallel Prediction Algorithm

```
1. start
2. Initialize MPI states
3. If rank of the process is 0 do steps 4-29 until the Robot moves in the environment
4. Begin process 0
5.  Get_Sensor_Reading(Obj1_t1)
6.  Get_Sensor_Reading(Obj2_t1)
7.  Get_Sensor_Reading(Objn_t1)
8. Get_Sensor_Reading(Obj1_t2)
9. Get_Sensor_Reading(Obj2_t2)
10. .......
11. Get_Sensor_Reading(Objn_t2)
```

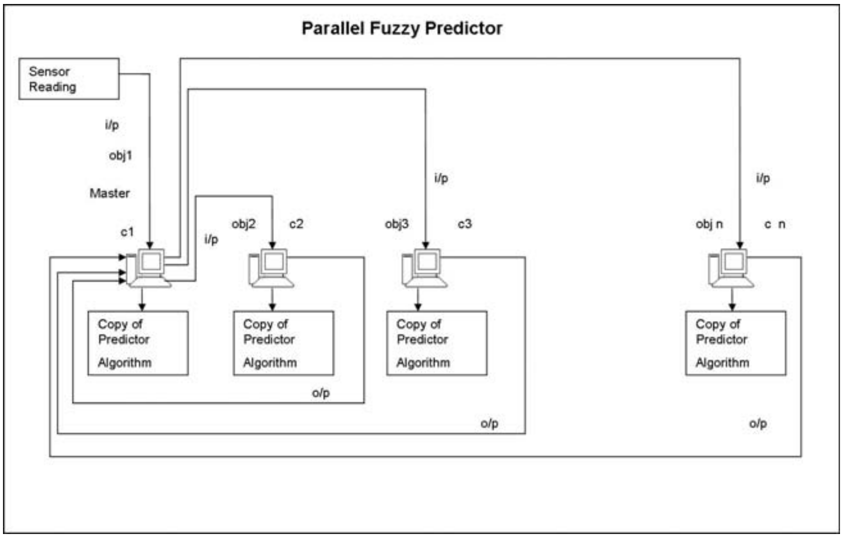

Parallel Fuzzy Predictor
Sensor Reading
i/p
obj1
Master
c1
i/p
obj2
c2
obj3
c3
i/p
obj n
c n
i/p
Copy of Predictor Algorithm
Copy of Predictor Algorithm
Copy of Predictor Algorithm
Copy of Predictor Algorithm
o/p
o/p
o/p

Fig. 1. Object Motion Prediction on a Cluster Computing System

```
12. if object 2 has not reached destination then Send object2 data to the process
    with Rank1
13. if object_n has not reached destination then Send object_n data to the proces-
    sor with Rank n-1
14. if object 1 has not reached destination then Call Short_term_predictor(Obj1_t1,
    Obj1_t2, return Obj1_t3)
15. if object 1 has not reached destination then Call Long_term_predictor(Obj1_t1,
    Obj1_t2, return winner_index)
16. Receive data from process2
17. .......
18. Receive data from process n
19. Obj1_t1=Obj1_t2
20. Obj2_t1=Obj2_t2
21. .......
22. Objn_t1=Objn_t2
23. Decide the next action
24. if object1 has not reached destination then Get_Sensor_Reading(Obj1_t2) Goto
    step 8
25. if object2 has not reached destination then Get_Sensor_Reading(Obj2_t2)Goto
    step 9
26. .......
27. if object_n has not reached destination then Get_Sensor_Reading(Objn_t2)Goto
    step 11
28. Finalize MPI states
29. End Processor 0
```

```
30. If (process_id=1) do steps 31-35 till the object_2 reaches destination
31. Receieve data of object_2 from processor0
32. Call Short_term_predictor(Obj2_t1,Obj2_t2, return Obj2_t3)
33. Call Long_term_predictor(Obj2_t1,Obj2_t2, return winner_index)
34. Send the results of prediction to Rank 0
35. End Processor 1
36. .......
37. If (process_id=n) do steps 38-42 till the object n reaches destination
38. Receieve data of object_n from processor_0
39. Call Short_term_predictor(Objn_t1,Objn_t2, return Objn_t3)
40. Call Long_term_predictor(Objn_t1,Objn_t2, return winner_index)
41. Send the results of prediction to Rank 0
42. End Processor n
43. End
```

3.2 Time Complexity

Let p be the number of processors in the cluster. The complexity of the algorithm is $((N+M)/p)$,where N is the time complexity of short term prediction and M is the time complexity of the Long term prediction. Passing a message of length n from one processor to another has time complexity $\Theta(n)$. Since broadcasting to p processors require $[\log p]$ message passing steps, the overall time complexity of broadcasting each iteration is $\Theta(n \log p)$.

The parallel program requires n broadcasts . Each broadcast has $[\log p]$ steps. Each step involves passing messages that are n bytes long. Hence the expected communication time of parallel program is

$$[n\,[\log p](\,\lambda + n/\beta)] \tag{2}$$

Adding computation time to communication time gives

$$[(N+M)/p + n\,[\log p](\,\lambda + n/\beta)] \tag{3}$$

However this expression will over estimate the parallel execution time because it ignores the fact that there can be considerable overlap between computation and communication.The message transmission time after first iteration is completely overlapped by computation time and should not be counted towards total execution time.Expected execution time

$$[(N+M)/p + n[\log p]\lambda + [\log p](n/\beta)] \tag{4}$$

4 Experimental Results and Analysis

We have performed experiments with both real and simulated data.Real data has been gathered through bench mark data sets available on line from i)INRIA Labs with data captured at INRIA Labs at Grenoble, France,ii)From Motion Capture

Web group of Univ of S.California and iii) From CMU Graphics Lab dataset. The data sets consist of different human motion patterns. Simulated data consists of noisy trajectories between predefined sequence of control points. Cluster computing is setup using three IBM Intellistation Intel Xeon machines (processor speed at 2.66 GHz). The parallelization of the algorithm is done using Message Passing Interface (MPI) in C++ environment. The predictor algorithm is run in parallel to keep track of multiple number of objects simultaneously(Figure2). Each node in the cluster keeps track of a unique object observed in the scene. To test the performance of the cluster system, the parallel predictor algorithm is run on 02 and 03 processors (for 02 and 03 objects) with similar test cases.

Some of the results obtained for both short term and long term prediction are shown in Figure 2. Each Sub Figure represents the top view of the human motions observed in the environment. Results of both short term and Long term motion predictions are shown separately to make the representations clear. Figure 2 a,b represent the movement of the object from left to right direction and the corresponding short term motion prediction path. Pi and Ai represent the predicted and actual path traversed by the moving object. Pi(G) and Ai(G) represent Predicted and actual goal of the object. A1 is the actual path observed and A1(G) is the actual goal reached by the object A1. Figure 2a represents a typical scenario when two moving objects were observed and 2b represents another scenario when three moving objects were observed.

Figure2c,d represent the Long term motion prediction results when two and three objects are in observation. Ai is the actual path observed and Ai(G) is the actual goal reached. In the initial stage based on the partial trajectory observed for the one of the moving objects(A1(G)), our predictor assumes P1 as the predicted trajectory and P1(G) as the final destination (in this case it is the outside door). But after reaching the intermediate goal (junction of multiple trajectories) , the object decides to move in the direction of another destination. Our predictor immediately switches itself in the new direction and predicts trajectory P1_2 and goal as P1_2(G). Similar observation is made for another object (A2(G)) in the scene. In Figure 2d three moving objects are considered and their corresponding predicted trajectories are shown. Performance of the proposed parallel algorithm, in terms of speedup and efficiency , on two and three processors is given in Table 1 and Table 2.

In order to run any problem in parallel we can check how much speed up achievable by our algorithm when run in parallel. Some basic principles need to be considered.The lower bound on the speedup $log_2 n$ on a n processor system is known as Minsky's conjecture. This is pessimistic approach. A more optimistic speedup estimate for upper bound can be given by Amdahl's law. Let f be the fraction of operations in a computation that must be performed sequentially , where $0 <= f <= 1$. The maximum speedup achievable by a parallel computer with p processors performing the computation is

$$\psi <= 1/(f + (1 - f)/p) \tag{5}$$

Amdahl's law is based on the assumption that we are trying to solve a problem of fixed size as quickly as possible . It provides an upper bound on the speedup

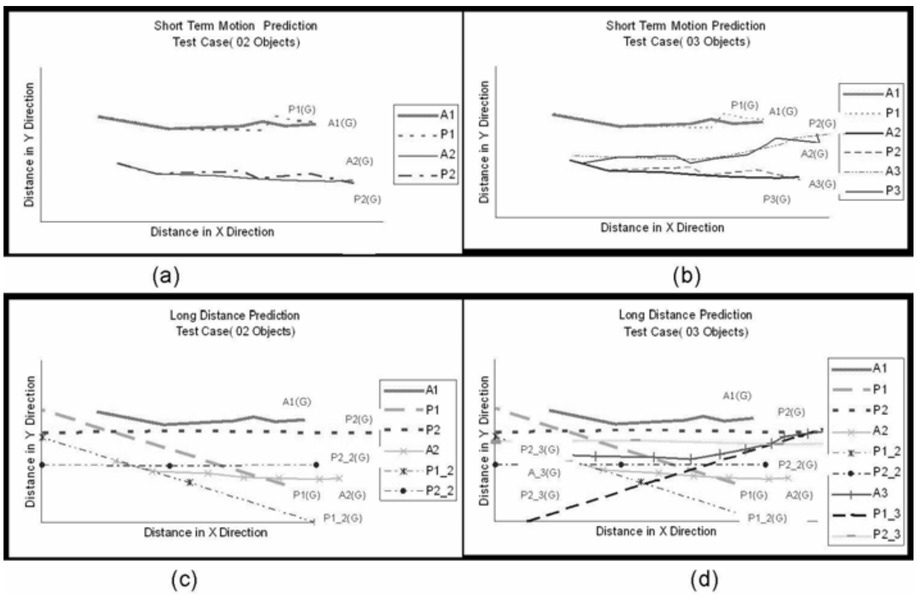


Fig. 2. Prediction graphs showing parallel implementation of (a)(b) Short Term motion prediction on 02 and 03 objects (c)(d)Long Term motion prediction on 02 and 03 objects

Table 1. Speedup and Efficiency of Parallel Motion Prediction algorithm for two processors

serial (02 objects)in Microseconds	parallel (2p)in Microseconds	Speedup	Efficiency
444	260	1.70	0.85
410	250	1.64	0.82
370	253	1.46	0.73
386	248	1.55	0.77
360	247	1.45	0.72
363	242	1.5	0.75
389	248	1.56	0.78
301	185	1.62	0.81
392	251	1.56	0.78
461	258	1.78	0.89
421	258	1.63	0.81
509	306	1.66	0.83
562	306	1.83	0.91

Table 2. Speedup and Efficiency of Parallel Motion Prediction algorithm for three processors

Serial (03 objects)in Microseconds	Parallel (3p)in Microseconds	Speedup	Efficiency
594	234	2.53	0.84
476	188	2.53	0.84
564	244	2.31	0.77
528	225	2.34	0.78
756	306	2.47	0.82
642	283	2.26	0.75
781	300	2.60	0.86
739	315	2.34	0.78
729	275	2.65	0.88
626	244	2.56	0.85
726	300	2.42	0.80
689	263	2.61	0.87
702	279	2.51	0.83

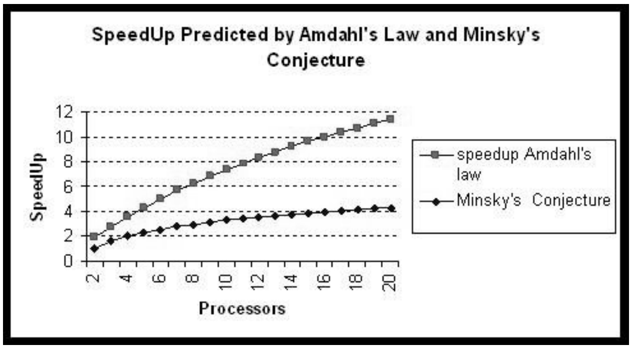


Fig. 3. Speedup Predicted by Amdahl's Law and Minsky's conjecture

achievable by applying a certain number of processors to solve the problem in parallel. In the proposed parallel prediction algorithm the number of instructions which are strictly sequential is 4% of the complete algorithm.These instructions include initialization of message passing instructions, getting sensor readings for each object and getting results from all cluster nodes. Figure 3 gives the expected performance of the proposed approach as per Amdahl's law and Minsky's conjecture [8]. From the graph it can be observed that the performance of the algorithm improves in the initial stage with increase in number of processors and becomes constant though there is increase in number of processors after reaching certain threshold.

Table 3. Comparison of Speedup of Parallel prediction algorithm with Amdahl's law and Minsky's Conjecture

Processors	Average Speedup (On Cluster)	Speedup (Amdahl's law)	Speedup (Minsky's conjecture)
2	1.57	1.92	1
3	2.21	2.77	1.58

Average speedup achieved on two and three processors in the cluster, by the proposed parallel predictor, are compared with the predictions of Amdahl's law and Minsky's conjecture(Table 3). The performance of the algorithm is better than the Minsky's conjecture , still less than the values predicted by Amdahl's law. It can be concluded from our experiments that there is limitation on number of processors which can be added to the cluster for parallel motion prediction and parallel implementation will reach its threshold earlier than predicted by the Amdahl's law. The reduction in prediction time can be attributed to communication delay and architectural constraints of the cluster system.

5 Conclusion

In this paper we have proposed a parallel motion prediction algorithm, for predicting the motion of multiple dynamic objects, in Robotic navigational environment simultaneously. Both Short term and Long term motion prediction techniques are incorporated in the proposed parallel prediction algorithm. Results of the study indicate that the parallel implementation of prediction algorithms improve the speed up of the prediction time considerably. However the increase in speedup is not linear due to the strictly sequential instructions present in the algorithm and communication delay in the cluster setup, which puts a limitation on the number of nodes in the cluster setup. Future work includes the efficient decision making regarding Robotś future move,based on the predicted outputs received from different nodes of the cluster.

Acknowledgements

The authors are indebted to AICTE Govt of India for funding the project.The authors are thankful to the benchmark dataset provided by EC Funded CAVIAR project, CMU Graphics lab and Motion capture web group.

References

1. Foka, A., Trahanias, P.E.: Predictive Autonomous Robot navigation. In: Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems, EFPL, Lausanne, Switzerland, October 2002, pp. 490–494 (2002)

2. Messom, C.H., Sen Gupta, G., Demidenko, S., Siong, L.Y.: Improving predictive control of a mobile robot: Application of image processing and kalman filtering. In: IMTC 2003 Instrumentation and Measurement Technology Conference, Vail, CO, USA, May 2003, pp. 20–22 (2003)
3. Perez, C., Reinoso, O., Vicente, M.A.: Robot hand visual tracking using an adaptive fuzzy logic controller. In: Proceedings of WSCGS 2004, Plzen, Czech Republic (February 2004)
4. Aycard, O., Petti, S., Vasquez, A.D., Yguel, M., Fraichard, T., Aycard, O.: Steps Towards Safe Navigation in Open and Dynamic Environments. In: Laugier, C., Chatila, R. (eds.) Autonomous Navigation in Dynamic Environments: Models and Algorithms. Springer Tracts in Advanced Robotics Series (STAR). Springer, Heidelberg (2006)
5. Henrich, D.: Fast Motion Planning by Parallel Processing – a Review. Journal of Intelligent and Robotic Systems 20, 45–69 (1997)
6. Mazer, E., Ahuactzin, J.M., Talbi, E.-G., Bessiére, P., Chatroux, T.: Parallel Motion Planning with the Ariadne's Clew Algorithm. Lecture Notes in Control and Information Sciences, vol. 200, archive, pp. 62–74. Springer, London (1993)
7. Yu, H., Su, T.: A Destination Driven Navigator with Dynamic Object Motion Prediction. In: The Proceedings of International Conference on Robotics 2L Automation, Seoul, Korea, May 21-26, pp. 2692–2697 (2001)
8. Hwang, K., Briggs, F.A.: Computer Architecture and Parallel Processing. McGraw-Hill, New York (1985)
9. Seyr, M., Jakubek, S., Novak, G.: Neural network predictive trajectory tracking of an autonomous two-wheeled mobile robot. In: Proceedings of IFAC World Congress (2005)
10. Caselli, S., Reggiani, M., Sbravati, R.: Parallel Path Planning with Multiple Evasion Strategies. In: Proceedings of the 2002 IEEE International Conference on Robotics & Automation Washington, DC, May 2002, pp. 260–266 (2002)
11. Rajpurohit, V.S., Manohara Pai, M.M.: An Optimized Fuzzy Based Short Term Object Motion Prediction for Real-Life Robot Navigation Environment. In: Sebillo, M., Vitiello, G., Schaefer, G. (eds.) VISUAL 2008. LNCS, vol. 5188, pp. 114–125. Springer, Heidelberg (2008)
12. Rajpurohit, V.S., Manohara Pai, M.M.: Using Self Organizing Networks for Moving Object Trajectory Prediction. International Journal on Artificial Intelligence and Machine Learning 9(1), 27–34 (2009)

Numerical Simulations of Unsteady Shock Wave Interactions Using SaC and Fortran-90

Daniel Rolls[2], Carl Joslin[2], Alexei Kudryavtsev[1], Sven-Bodo Scholz[2], and Alex Shafarenko[2]

[1] Institute of Theoretical and Applied Mechanics RAS SB,
Institutskaya st. 4/5,
Novosibirsk, 630090, Russia
[2] Department of Computer Science, University of Hertfordshire, AL10 9AB, UK

Abstract. This paper briefly introduces SaC: a data-parallel language with an imperative feel but side-effect free and declarative. The experiences of porting a simulation of unsteady shock waves in the Euler system from Fortran to SaC are reported. Both the SaC and Fortran code was run on a 16-core AMD machine. We demonstrate scalability and performance of our approach by comparison to Fortran.

1 Introduction

In the past when high performance was desired from code, high-levels of abstraction had to be comprimised. This paper will demonstrate our approach which overcomes these shortcomings: we will present the data-parallel language SaC [14] and exemplify its usage by implementing an unsteady shock wave simulator in the Euler system. SaC was developed by an international consortium coordinated by one of the authors (Sven-Bodo Scholz). We will compare the performance of our approach against Fortran by running this application on a 16-core computation server.

The language is close to C syntactically, which makes it more accessible to computational scientists, while at the same time being a side-effect free, declarative language. The latter enables a whole host of intricate optimisations in the compiler and, perhaps more importantly, liberates the programmer from implementation concerns, such as the efficiency of memory access and space management, exploitation of data-parallelism and optimisation of iteration spaces. In addition, code that was written for a specific dimensionality of arrays can be reused in higher dimensions thanks to an elaborate system of array subtyping in SaC, as well as its facilities for function and operator overloading that far exceed the capabilities of not only Fortran but the object-orientation languages as well.

SaC has already been used for many kinds of application, ranging from image-processing to cryptography to signal analysis. However, to our knowledge there has been only one occasion of programming a Computational Fluid Dymamics application in SaC namely the Kademtsev-Petviashivili system [4]. Even that example is too esoteric to support any conclusions about practical suitability of Single-Assignment C. In this paper we present for the first time the results of using SaC as a tool in solving a real, practical problem: simulation of unsteady shock waves in the Euler system.

The equations of fluid mechanics can be solved analitically for only a limited number of simple flows. As a consequence, numerical simulation of fluid flows known as Computational Fluid Dynamics (CFD) is widely used in both scientific research and countless engineering applications. Efficiency of computations and ease of code development is of great importance in CFD which is one of the most perspective fields for implementing new concepts and tools of computer science.

In Section 2 we will briefly outline the features of SaC that we would argue make it uniquely suitable for the class of applications being discussed. Section 3 delineates the numerical method being used and Section 4 discuses implementation issues we came across when porting a Fortran TVD implementatin to SaC. Our results are then presented in Section 5 and related work is discussed in Section 6 before finally Section 7 discusses the lessons learnt and concludes.

2 SaC

SaC is an array processing language that first appears to be an imperitive program like Fortran but actually has more in common with functional programming languages. A SaC function consists of a sequence of statements that define and re-define array objects. To a C programmer this looks very similar to assigning the result of expressions to arrays, but there is an important difference: what may appear to the programmer to be the "control flow" in SaC is in fact a chain of definitions that link with one another via the use of common variables, this emphasises data as opposed to control dependencies. Thus any iterative update becomes essentially a recurrence relation between the snapshots of the arrays being updated, and it is up to the compiler whether or not the arrays need to be recreated as objects in memory or whether the underlying computation may be taken in-flow. That not withstanding, analogues of control structures, such as the `IF` statement, are provided, if only with a slightly different interpretation, so the illusion of programming a control flow may be retained as far as possible. `IF` statements are expressions: this can be seen by observing that with imperitive code, control flow through conditionals can affect whether a variable is defined; however this is not valid SaC code.

Two main constructs of SaC support the kind of computation that we are concerned with in this paper:

Most of the high level constructions in this paper are compiled down to the following to constructs.

with-loop. Despite the name, which reflects some historic choices of terminology in SAC, the essence of this construct is a data-parallel array definition. The programmer supplies a specification of the index space (in an extended enumeration form) and the definition of the array value for a given index in terms of an expression with other values possibly indexed and produced by external functions. Definitions for different array values are assumed to be mutually independent, hence data-parallelism is presented to the compiler explicitly.

for loop. This is used for programming recurrences. The recurrence index is specified in the for loop together with its initial value and increment, the compiler interprets the loop body as a definition of the arrays emerging at the final step of the recurrence in terms of the arrays defined prior to the first step.

As with FORTRAN-90 small arithmetic expressions in SAC can operate on whole arrays to conveniently express elementwise operations on those arrays. E.g. `a - b * c + c` could be both an expression operating on scalars, arrays or scalars and arrays where the scalar form of the expression is applied to corresponding indicies in the arrays `a`, `b` and `c`. For consise expressivness SAC supports set notation which allows an expression to be defined for every element of a new array where each expression may depend on the index. E.g. `{ [i,j] -> matrix[j,i] }` transposes a matrix by placing element (j, i) from the original matrix into element (i, j) for all i and j.

Another feature of the language that finds its use in the application being reported is its type system, which supports subtyping. To provide an overview of this, we remark, by way of an example, that a vector can be interpreted as a two dimensional array obtained by replicating the vector as a row in the column dimension. This is a subtype of a general two dimensional array type. One consequence of this is that a function that contains a tridiagonal solver for a one-dimensional Poisson equation can be applied to a two dimensional array (acting row-wise) and then applied again column-wise by using two transpositions, all without changing a single line of code in the solver definition.

All these features make it possible to write function bodies that act on inputs of any dimension which suffer no performance loss compared to more specialized function bodies. Our code makes use of this fact to reuse function bodies for a one dimensional and two dimensional shockwave simulation.

3 Application

SAC is used to develop an efficient solver for the compressible Euler equations, which govern the flow of an inviscid perfect gas:

$$\frac{\partial \mathbf{Q}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = 0, \tag{1}$$

$$\mathbf{Q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}, \qquad \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 \\ \rho uv \\ u(E+p) \end{bmatrix}, \qquad \mathbf{G} = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 \\ v(E+p) \end{bmatrix}. \tag{2}$$

Here t is time, x and y are spatial coordinates, u and v are components of the flow velocity, ρ is density, p is the pressure related to the total energy E as

$$p = (\gamma - 1)\left(E - \rho\frac{u^2 + v^2}{2}\right), \tag{3}$$

where γ is the ratio of specific heats ($\gamma = 1.4$ for air). The Euler equations are the canonical example of a hyperbolic system of nonlinear conservation laws that describes conservation of mass, momentum and energy. Numerical methods, originally developed for the Euler equations, can be also used for a wide variety of other hyperbolic systems of conservation laws, which arise in physical models describing physical phenomena in fields as varied as acoustics and gas dynamics, traffic flow, elasticity, astrophysics and cosmology. Thus, the Euler solver is a very representative example of a broad class of computational physics programs.

A salient feature of nonlinear hyperbolic equations is the emergence of discontinuous solutions such as shock waves, fluid and material interfaces. It turns their numerical solution into a non-trivial task. Modern numerical methods for solving the hyperbolic equation [9] are based on high-resolution shock-capturing schemes originated from the seminal Godunov's paper [7]. In these methods, the computational domain is divided into a number of grid cells and the conservation laws are written for each cell. The computational procedure includes three stages: 1) reconstruction (in each cell) of the flow variables on the cell faces from cell-averaged variables; 2) evaluation of the numerical fluxes through the cell boundaries; and 3) advancement of the solution from the time t^n to time t^{n+1} where $t^{n+1} = t^n + \Delta t$. These stages are successively reiterated during the time intergation of Eq (1).

The reconstruction during the first stage should avoid the interpolation across the flow discontinuities. Otherwise, numerical simulations fail because of a loss of monotonicity and numerical oscillations developing near the discontinuities. The Fortran code developed includes several techniques of monotone reconstruction, in particular, the TVD (Total Variation Diminishing) reconstructions of the 2nd and 3rd orders with various slope limiters and the 3rd order WENO (Weighted Essentially Non-Oscillatory) reconstruction, which automatically assigns the zero weight to the stencils crossing a discontinuity. The latter technique is used in the examples of flow computation below. The reconstruction is applied to the so-called (local) characterisic variables rather than to the primitive variables ρ, u, v and p or the conservative variables $\mathbf{Q}$.

The evaluation of numerical axes is performed by approximately solving the Riemann problems between two states on the "left" and "right" sides of the cell boundaries resulting from the reconstruction. The code includes a few options for the approximate Riemann solver, below the results obtained from the shock wave simulation are presented. For time advancement (Stage 3) the 2nd or 3rd order TVD Runge-Kutta schemes are used.

As an example of flow computations both a one dimensional and two dimensional problem is described below.

3.1 One Dimensional Simulation

The Euler code was used to solve the Sod shock tube problem [16], a common test for the accuracy of computational gasdynamics code. The test consists of a one dimensional Riemann problem. At the initial moment, the diaphragm separates two resting gases with different pressures and densities. The top state is $(\rho, u, p) = (1, 0, 1)$ while the bottom state is $(\rho, u, p) = (0.125, 0, 0.1)$. Here ρ is the density, u is the flow velocity and p is the pressure. After the diaphragm rapture, a shock wave and a contact discontinuity propagates to the bottom and a rarefaction wave moves to the top. This is illustrated in Fig. 1.

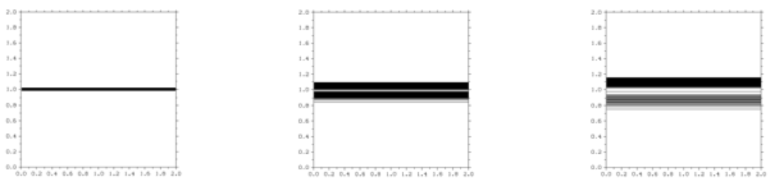


Fig. 1. The expansion of a shockwave from the center in the one-dimensional simulation where two gasses of different densities meet. The three diagrams move forward in time from left to right and show the shockwave expanding.

3.2 Two-Dimensional Simulation

Here a numerical simulation of an unsteady shock wave interaction is conducted. A schematic of flow configuration is shown in Fig. 2. The computational domain is a square divided into rectangular grid of $N_x \times N_y$ cells. A part of its left boundary is the exit section of a channel while the remaining portion of this boundary is a solid wall. The exit section of another channel comprises part of the computational domain's bottom boundary. A shock waves propagates within each of the channels and comes to the channels exits at the same moment ($t = 0$) when the computation starta. Thus, at the initial moment, the domain is filled by a quiescent gas. The boundary conditions in the exit sections of two channels are imposed in such a way that the flow variables are equal to the values behind the shock waves calculated from the Rankine-Hugoniot relations.

The computations have been conducted at the shock wave Mach numbers of $M_s = 2.2$. At this value of M_s the flow behind the shock waves is supersonic so that the flow variables in the exit sections are not changed during the computation. The size of the computational domain is $L_x = L_y = 2h$, where h is the channel width and $h = 200$ in our benchmarks.

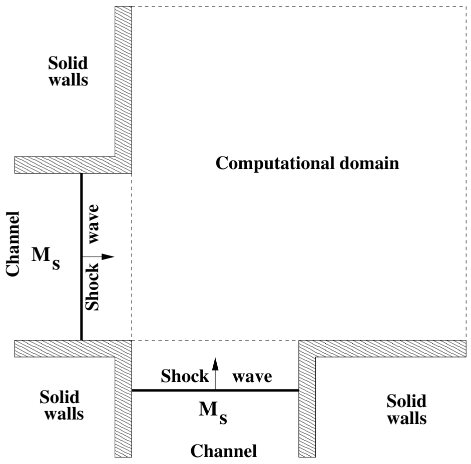


Fig. 2. A schematic of flow configuration and computational domain for the two-dimensional simulation

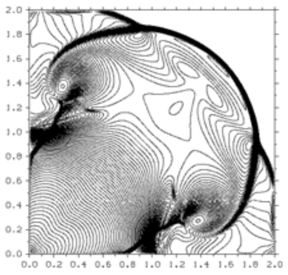


Fig. 3. A snapshot of the shockwave in the two-dimensional simulation

The results of computations are shown in Fig. 3. The interaction between the shock waves exhausting from the channels and their diffraction over solid walls generate a complex flow structure. In addition to the primary shock waves, which rapidly become approximately circular in shape, the irregular interaction of the shock waves leads to formation of a Mach stem between them and emergence of two reflected shock waves. The primary wave, the relected shock wave and the Mach stem meet in the three points, from which slipstream surfaces emanate. Behind each of the primary shock waves, there is a contact surface separating the gas exhausted out of the channel from the gas which initially filled the computatational domain. Secondary shock waves are formed closer to the exit section starting from a point on the last characteristics of the channel lips. On the later stages of evolution, the Mach stem itself becomes circular in shape and occupies a large proportion of the leading shock front while the contact surface behind it curls up into a mushroom-like structure.

4 Implementation

To illustrate the arguments from Section 2 we have selected two example functions from our TVD implementation in SaC.

4.1 dfDxNoBoundary

The function `dfDxNoBoundary` produces an array of the difference between each neighbouring pair in a vector. It takes the difference of every element in a vector but its first element with its left-neighbouring element and divides each element by a constant. The resulting vector has a length of one element less than the input vector.

As with the Fortran, in SaC the original vector is extended on both ends. The function defines two new vectors, one with the first element removed and one with the last element removed. An element-wise subtraction is applied to these new vectors (with matching indexes) and the resulting array is divided elementwise by a scalar (delta).

```
inline
fluid_cv[.] dfDxNoBoundary( fluid_cv[.] dqc, double delta)
{
  return( ( drop([1], dqc) - drop( [-1], dqc) ) / delta);
}
```

To materialise each array in memory would be expensive; this style of programming would not be feasible for computational science if every array was copied. SaC's functional underpinnings allow it to, among other things, avoid some unnecessary calculations, memory allocation and memory copies. The style of code above often performs extremely well contrary to initial expectations.

4.2 getDT

The `GetDT` function calculates the time step to take in each iteration of the algorithm. It acts upon every element in a large array which represents the computational domain. For the two dimensional case Fortran has a nested loop structure with one loop for each dimension. The value EV is calculated each time and the largest EV value is saved. Finally this value is divided by a constant.

```
      SUBROUTINE GetDT
      USE Cons
      USE Vars
      IMPLICIT REAL*8 (A-H,O-Z)

      EVmax = 0.d0
      DO iy=IYmin,IYmax
      DO ix=IXmin,IXmax
        Ux = QP(1,ix,iy)
        Uy = QP(2,ix,iy)
        Pc = QP(3,ix,iy)
```

```
          Rc = QP(4,ix,iy)
          C = SQRT(Gam*Pc/Rc)
          EV = (ABS(Ux)+C)/Dx+(ABS(Uy)+C)/Dy
          EVmax = MAX(EV,EVmax)
        END DO
        END DO

        DT = CFL/EVmax

        END
```

The SaC version of the function is shown below. In the following code GAM, DELTA and CFL are constants.

```
1 inline
2 double getDt(fluid_pv[+] qp)
3 {
4   c = sqrt(GAM * p(qp) / rho(qp));
5   d = MathArray::fabs( u(qp));
6   ev = { iv -> (sum( ( d[iv] + c[iv]) / DELTA))};
7   return( CFL / maxval( ev));
8 }
```

The type of the function parameter is fluid_pv[+] which means an array of unknown dimensionality of fluid_pv values where fluid_pv is a user defined datatype. The syntax for an array type (t) can be syntactically represented as t[x,y,z] for an array of size x by y by z, t[.,.] for a array of two dimensions of unknown size and also t[+] for an array of unknown dimensionality.

The functions p and ρ extract the pressure and density from fluid_pv respectively. The SaC function calculates the variable C above using elementwise operations and then in line 6 EV is calculated which depends on the entire input array. With little experience with SaC this function quickly becomes easier to understand than the Fortran code. It is a functional definition (i.e. an expression) but the programmer is not obliged to use recursion on the array like a functional programmer would do with lists.

This clearer imperative-like but functional style makes data dependencies more obvious both to the programmer and to the compiler. In our simulation the SaC compiler always calculates the dimensionality needed for this function from its calls and therefore no penalty is paid for the generic type of qp.

5 Results

To evaluate the performance of SaC compared with Fortran we ran the 2D simulation with a 400x400 grid as described in Section 3.2. The simulation was run for 1000 time steps to ensure that the run time was sufficient to negate the start-up time of the program. We made use of a 400x400 grid as this was the size used in the original Fortran implementation. In the experiment we used the third order Runge-Kutta TVD method and first order piecewise constant reconstruction.

Compiler	Version	Arguments
SAC2C	SAC2C 16094 stdlib 1120	-L fluid -maxoptcyc 100 -O3 -mt -DDIM=2 -nofoldparallel -maxwlur 20
Sun Studio Compiler-f90	8.3 Linux i386 Patch 127145-01	-autopar -parallel -loopinfo -reduction -O3 -fast

The computer used to perform these benchmarks is a 4xQuad-Core (16 core) AMD Opteron^{TM} 8356 with 16GB of RAM. The source code is available at http://sac-home.org.

As the Fortran compiler uses OpenMP for parallelization, environment variables where set to control the runtime behaver of the Fortran code. Several different combinations where tried however these made a negligible difference to the runtime of the program. The options that produced the fastest runtimes, and therefore where used for the main benchmarking, were: `OMP_SCHEDULE=STATIC`, `OMP_NESTED=TRUE` and `OMP_DYNAMIC=FALSE`.

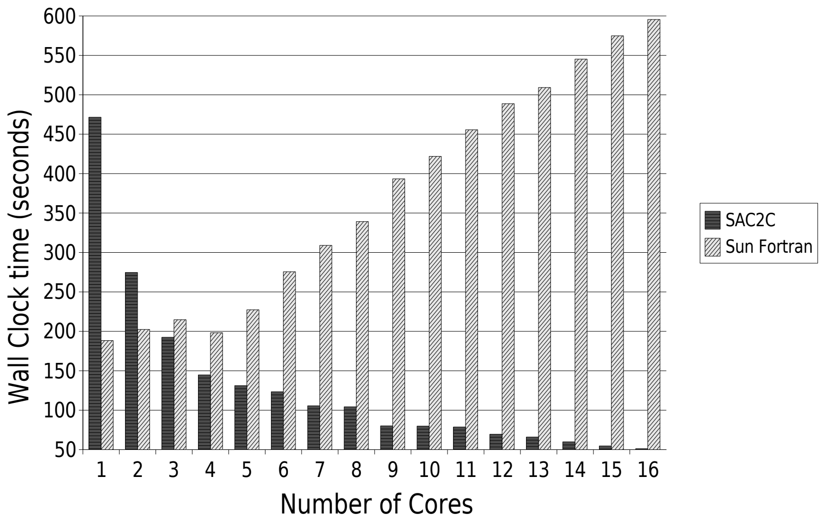


Fig. 4. Wall clock time of a 1000 time step simulation on a 400x400 grid

It can be seen in Figure 4 that SAC was much slower than the Fortran when run on just one core. However the Fortran code did not scale well with the number of cores, and as the number of cores increased performance degraded. We therefore suspect that there is added overhead of communication between the threads.

SAC does not use system calls for its inter thread communication but rather uses the programs shared memory and spin locks to allow inter thread communication with very little overhead. This low overhead allows SAC to scale well even when its problem size is to small for Fortran's auto parallelize feature to work efficiently. There are optimizations which the SaC compiler can perform which are only possible because SaC is a functional, single assignment language. These optimizations help to allow the program to scale as SaC collates the many small operations on the arrays into fewer larger operations. This is not possible in procedural programing languages like Fortran as the compiler can not always work out the data dependences in complete detail. With a functional programing language like SaC it is possible to identify every dependency.

When the same benchmark was run with a larger 2000x2000 grid we discovered that Fortran was able to scale slightly with small numbers of cores but after just five cores it started to suffer from the overheads of inter-thread communication again.

6 Related Work

Broadly three techniques exist for producing highly parallelizable code for scientific simulations. The first technique is to carefully determine how a run should be parallelised and to explicitly write the code to do this. The message passing interface API[6] is commonly used for this. Also a threading library like Pthreads [11] could be used. Secondly, source-code annotations or directives can be used to provide information to a compiler to show it how an execution can be parallelised. Lastly compilers can try to autoparallelize code by analysing dependencies between variables. This section gives a brief overview of the three methods mentioned above and then discusses performance.

High-Performance Fortran [5] is an extension to FORTRAN-90 that allows the addition of directives to the source code to annotate distribution and locality. The Fortran code itself is written in a sequential style and already describes some operations in a data-parallel way. High-Performance Fortran compilers can then use these directives to compile to pipelined, vectorized, SIMD or message passing parallel code.

For explicitly annotating parts of a program that can be parallelized on shared memory systems the OpenMP [3] API is supported for C, C++ and Fortran. Many autoparallelizing compilers produce programs that call upon this API including the Intel and Sun Microsystems Fortran and C compilers.

ZPL [12] is a high level array processing language designed to be concise and platform independent. It allows programmers to easily describe subarrays within an array using a concept called regions. ZPL was designed with parallelism in mind and has had its performance compared with other languages for applications inclusive of computational fluid dynamics applications [13].

Parallel performance results tend to vary depending on the application, architecture and type of parallelism. For example an application that performs well on shared memory systems may not necessarily perform well when compiled to run on a distributed memory system. In addition and rather surprisingly, carefully crafted MPI applications might not necessarily have better speedups per core than implicit parallelism in high level languages. One surprising example of this is ZPL which has been shown to scale well in parallel runs with the Multigrid [1] NAS benchmark [2] and even shown prospects with CFD [13].

7 Conclusion

The results have shown that execution of high-performance applications written in SaC can achieve speedups on parallel architectures. This shows that using high-level abstractions in code operating on arrays is easier to understand. However, SaC's real strength comes into play when auto-parallelizing code.

SaC provides a powerful expressiveness where the greater learning curve is not grasping the paradigm but in resisting the temptation to try to optimize the code, and thus making use of SaC's ability to allow programs to be written with a high level of abstraction.

Programmers need a good way to express their programs so that they are quick to write, easy to understand and efficient to maintain. Up until now expressing programs in a readable form, with high levels of abstraction has come at a considerable performance penalty. However as can be seen in this paper it is possible to write programs in a clear style with high levels of abstraction while obtaining reasonable speedups that can be greater than those produced from the compilers of languages that where originally designed as sequential languages.

The results of this paper used SaC's Pthread back-end; future SaC back-ends promise even more parallelism. CUDA [10], whilst challenging to harness, has tremendous processing capabilities that will enable programs to make use of high performance, low cost processing resources found on GPUs as a potentially faster way of performing complicated simulations [15]. As part of an EU FP-7 project a back-end is being developed for SaC which produces code for an language which will compile to a many-core architecture called Microgrid [8]. This architecture will deliver considerable parallelism without the complexity that is involved with CUDA.

For future architectures parallelism will be increasingly vital. The work in this paper has shown that now that parallelism is important, it is possible to write abstract code in a high-level language and still be able to compete with traditional low-level, high performance languages like Fortran on parallel architectures.

Acknowledgments. This work was partly supported by the European FP-7 Integrated Project Apple-core (FP7-215216 — Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs).

References

1. Briggs, W.L., McCormick, S.F.: A multigrid tutorial. Society for Industrial Mathematics (2000)
2. Chamberlain, B.L., Deitz, S.J., Snyder, L.: A comparative study of the NAS MG benchmark across parallel languages and architectures. In: Supercomputing, ACM/IEEE 2000 Conference, pp. 46–46 (2000)
3. Chapman, B., Jost, G., Van Der Pas, R., Kuck, D.J.: Using OpenMP: portable shared memory parallel programming. The MIT Press, Cambridge (2007)

4. Shafarenko, A., et al.: Implementing a numerical solution of the kpi equation using single assignment c: Lessons learned and experiences. In: Implementation and Application of Functional Languages, 20th international symposium, pp. 160–170 (2005)
5. High Performance Fortran Forum. High Performance Fortran Language Specification. Rice University (1993)
6. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.1. High Performance Computing Center Stuttgart (HLRS) (2008)
7. Godunov, S.K.: A difference method for numerical calculation of discontinuous equations of hydrodynamics (in russian). Mat. Sb. 47, 271–300 (1959)
8. Grelck, C., Herhut, S., Jesshope, C., Joslin, C., Lankamp, M., Scholz, S.-B., Shafarenko, A.: Compiling the Functional Data-Parallel Language sac for Microgrids of Self-Adaptive Virtual Processors. In: 14th Workshop on Compilers for Parallel Computing (CPC 2009), IBM Research Center, Zurich, Switzerland (2009)
9. Guinot, V.: Godunov-type schemes. Elsevier, Amsterdam (2003)
10. Guo, J., Thiyagalingam, J., Scholz, S.-B.: Towards Compiling SAC to CUDA. In: Proceedings of the 10th Symposium On Trends In Functional Programming, Komarno, Slovakia (June 2009)
11. Josey, A.: The Single UNIX Specification Version 3. Open Group (2004)
12. Lin, C., Snyder, L.: ZPL: An array sublanguage. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1993. LNCS, vol. 768, pp. 96–114. Springer, Heidelberg (1994)
13. Lin, C., Snyder, L.: SIMPLE performance results in ZPL. In: Pingali, K.K., Gelernter, D., Padua, D.A., Banerjee, U., Nicolau, A. (eds.) LCPC 1994. LNCS, vol. 892, pp. 361–375. Springer, Heidelberg (1995)
14. Scholz, S.-B.: Single assignement c – efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13, 1005–1059 (2003)
15. Senocak, I., Thibault, J., Caylor, M.: J19. 2 Rapid-response Urban CFD Simulations using a GPU Computing Paradigm on Desktop Supercomputers
16. Sod, G.A.: A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. Journal of Computational Physics 27(1), 1–31 (1978)

Parallel Medical Image Reconstruction: From Graphics Processors to Grids

Maraike Schellmann, Sergei Gorlatch, Dominik Meiländer, Thomas Kösters, Klaus Schäfers, Frank Wübbeling, and Martin Burger

University of Münster, Germany
schellmann@uni-muenster.de

Abstract. We present a variety of possible parallelization approaches for a real-world case study using several modern parallel and distributed computer architectures. Our case study is a production-quality, time-intensive algorithm for medical image reconstruction used in computer tomography. We describe how this algorithm can be parallelized for the main kinds of contemporary parallel architectures: shared-memory multiprocessors, distributed-memory clusters, graphics processors, the Cell processor and, finally, how various architectures can be accessed in a distributed Grid environment. The main contribution of the paper, besides the parallelization approaches, is their systematic comparison regarding four important criteria: performance, programming comfort, accessibility, and cost-effectiveness. We report results of experiments on particular parallel machines of different architectures that confirm the findings of our systematic comparison.

1 Introduction

The research presented in this paper was conducted at the interdisciplinary collaborative research center (SFB) "Molecular Cardiovascular Imaging" at the University of Münster. One of the major medical research topics of the SFB is the detection of so-called vulnerable plaques in cardiac vessels using Positron Emission Tomography (PET) (a vulnerable plaque is an instable deposit within an arterial wall which may rupture and lead to a heart attack). In order to accurately detect such small-sized thickenings, it is necessary to have PET images with the highest spatial resolution.

The required improvement of images can be achieved by enhancing imaging hardware (scanners) or software (image reconstruction and processing algorithms). However, in either case, with increasing resolution and quality, the computational cost of the imaging algorithms increases, too. Today, the runtime on an off-the-shelf PC of one of the most accurate 3D PET reconstruction algorithms (the list-mode OSEM) used for reconstruction of data produced by a high-resolution small-animal PET scanner (the quadHIDAC [1]) ranges from one hour to several days. Therefore, parallelization is crucial in order for such hardware and software techniques to be used in clinical routine. With more advanced scanners and more precise and enhanced algorithms, an efficient parallel implementation will become even more important.

In this paper, we focus on the parallelization of the list-mode OSEM (Ordered Subset Expectation Maximization) algorithm [2] for PET reconstruction. The algorithm is representative for a large class of image reconstruction methods. The foremost goal of this work is to find the most suitable parallel architecture for this algorithm.

The suitability of a parallel architecture for a particular algorithm is usually defined using two criteria : 1) the algorithm's parallel performance, and 2) the usability of available programming environments for this particular architecture. In medical imaging, the medical personnel will be reluctant to use parallel software if it requires a high administrative effort like searching for a free time-slot on a number of cluster computers. Therefore, our third comparison criterion will be accessibility. Moreover, while a server with several multi-core processors might provide high performance and can be easily accessed over a local file system, the purchase cost of such a server might limit its usage. Thus, we introduce cost-effectiveness as our fourth criterion.

The contribution of this paper is three-fold: 1) It gives an overview of the parallel implementation of the list-mode OSEM algorithm on a number of parallel architectures, including: shared-memory multiprocessors, multi-core processors, cluster computers, graphics hardware and the Cell processor. 2) It identifies the most suitable parallel architecture for typical image reconstruction tasks by analyzing the parallel implementations for performance, appropriateness of programming environments, accessibility of the corresponding architecture and cost-effectiveness. 3) It describes how our grid system chooses the most suitable, currently available parallel machine for a given reconstruction task and thus frees the medical personnel from all administrative efforts.

The rest of the paper is structured as follows: we start with an introduction to PET image reconstruction and the list-mode OSEM algorithm in Section 2. We then introduce our parallel implementations on shared-memory machines, clusters, graphics hardware and the Cell processor in Section 3. In Section 4, we present runtime experiments and compare the parallel architectures using the four criteria introduced above. In Section 5, we describe the distributed MIRGrid system from a user perspective and, finally, present conclusions in Section 6.

2 Iterative PET Image Reconstruction

In Positron Emission Tomography (PET), a radioactive substance is injected into a human or animal body. Afterwards, the body is placed inside a PET scanner that contains several arrays of detectors. As the particles of the applied substance decay, positrons are emitted (hence the name PET) and annihilate with nearby electrons. During one such annihilation, two photons are emitted in opposite directions (see Fig. 1). The "decay events" are registered by two opposite detectors at the same time. The scanner records these events in a list with each record comprising the positions of those two detectors.

For our comparative study, we consider the following representative algorithm for creating an image from the events. List-Mode Ordered Subset Expectation

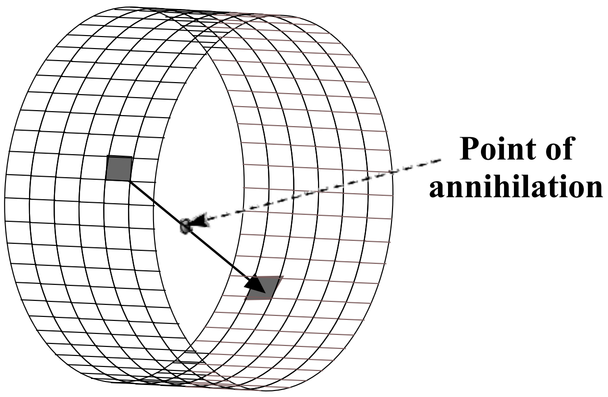


Fig. 1. Two detectors register an event in a PET-scanner

Maximization [2,3] (called list-mode OSEM in the sequel) is a block-iterative algorithm for 3D image reconstruction. List-mode OSEM takes a set of events and splits them into s equally sized subsets.
For each subset $l \in 0, \ldots, s-1$, the following computation is performed:

$$f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A_N^t \mathbf{1}} \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l}. \tag{1}$$

Here $f \in \mathbb{R}^n$ is a 3D image in vector form with dimensions $n = (X \times Y \times Z)$, $A \in \mathbb{R}^{m \times n}$, element a_{ik} of row A_i is the length of intersection of the line between the two detectors of event i with voxel k of the reconstruction region, computed with Siddon's algorithm [4]. $\frac{1}{A_N^t \mathbf{1}}$ is the so-called normalization vector. Since it can be precomputed, we will omit it in the following. Note that the multiplication of $f_l c_l$ is performed element-by-element. Each subset's computation takes its predecessor's output image as input and produces a new, more precise image.

The overall structure of a sequential list-mode OSEM implementation is shown in Listing 1. It comprises three nested loops, one outer loop with two inner loops. The outer loop iterates over the subsets. The first inner loop iterates over a subset's events to compute the summation part of c_l. The second inner loop iterates over all elements of f_l and c_l to compute f_{l+1}. The application in this study can be used to reconstruct data from virtually every PET scanner if a conversion method from the scanner data to world coordinates is available. In this study, we use data acquired by the quadHIDAC scanner and employ the conversion method for this scanner introduced in [2].

List-mode OSEM is a rather time-consuming algorithm. A typical 3D image reconstruction processing $6 \cdot 10^7$ input events for a $150 \times 150 \times 280$ PET image takes more than two hours on an off-the-shelf PC. To reduce the algorithm's runtime we developed several parallel implementations for systems with shared

```
for (int l = 0; l < subsets; l++) {
  /* read subset */

  /* compute c_l */
  for (int i = 0; i < subset_size; i++) {
    ...  }

  /* compute f_l+1 */
  for (int k = 0 ; k < image_size; k++) {
    if (c_l[k] > 0.0)
      f[k] *= c_l[k];
  }  }
```

Listing 1. Sequential code comprises one outer loop with two nested inner loops

and distributed memory, as well as hybrid systems [5,6]. An implementation for Compute Unified Device Architecture (CUDA) capable graphics processing units [7] as well as for the Cell processor [8] is also available.

In the following section, we systematically compare these implementations with respect to different criteria.

3 Parallel Image Reconstruction

Because of the data dependency between the subsets' computations in (1), implied by $f_{l+1} = f_l c_l$, the subsets cannot be processed in parallel. The computation of c_l and f_{l+1} is parallelizable, using the following idea.

For the computation of c_l, all parallel implementations distribute the events among the processing units (either processors or cores, from now on called PUs). Now each PU computes a partial sum of c_l. Afterwards, all partial results are summed up over the communication link. For the computation of $f_{l+1} = f_l c_l$, the image is distributed among the PUs, thus each PU computes $f_{l+1} = f_l c_l$ for its subimage in parallel.

3.1 Parallelization on Shared-Memory Processors

On shared-memory multiprocessors and multi-core processors, we developed an OpenMP [9] implementation following the parallelization idea described above.

To parallelize the computation of c_l and f_{l+1}, we have to parallelize the two inner loops of the list-mode OSEM algorithm. We use the `parallel for` directive of OpenMP that declares the successive for loop to be executed in parallel by a team of threads for both loops (see Listing 2). Apart from the additional compiler directives, no considerable changes were made to the sequential program. Thus, an OpenMP-based parallel implementation of the list-mode OSEM algorithm is easily derived from a sequential implementation.

```
for (int l = 0; l < subsets; l++) {
    /* read subset */
    /* compute c_l */
    #pragma omp parallel for
    for (int i = 0; i < subset_size; i++) {
        path=computePath(i);
        ...
        #pragma omp critical
        while (path[m].coord != -1) {
            c_l[path[m].coord]+=path[m].length*c;
        } /* end of critical section */
            ...
    }
    /* compute f_l+1 */
    #pragma omp parallel for
    for (int k = 0 ; k < image_size; k++) {
        if (c_l[k] > 0.0)
            f[k] = f[k] * c_l[k];  }  }
```

Listing 2. Sequential implementation with OpenMP compiler directives inserted

Within the first inner loop (summation part of c_l), all threads perform multiple additions to arbitrary voxels of a common intermediate image. We prevent race-conditions using a mutex that declares the summation part mutually exclusive, such that only one thread at a time is able to work on the image. In OpenMP, mutexes are declared by using the *critical* construct which specifies a mutual exclusion for the successive code section.

3.2 Parallelization on Cluster Computers

On distributed-memory clusters, we use MPI (Message Passing Interface) [10] for the parallel implementation. Here, every process first reads "its" events from the remote file system (see Fig. 2). All processes compute their partial sum of c_l simultaneously; then the result is summed up using `MPI_Allreduce`. Finally, before the next subset is started, all processes compute f_{l+1}. Note that for the computation of f_{l+1} the image is not distributed among the processes, because the resulting network communication is more time-consuming than the actual computations.

On hybrid machines (clusters), where each node is either a shared-memory multiprocessor or a multi-core processor, we combine the MPI distributed-memory implementation with the OpenMP shared-memory implementation. Thus the partial sum of c_l and f_{l+1} are computed simultaneously by all PUs of the shared-memory machines (see Fig. 2).

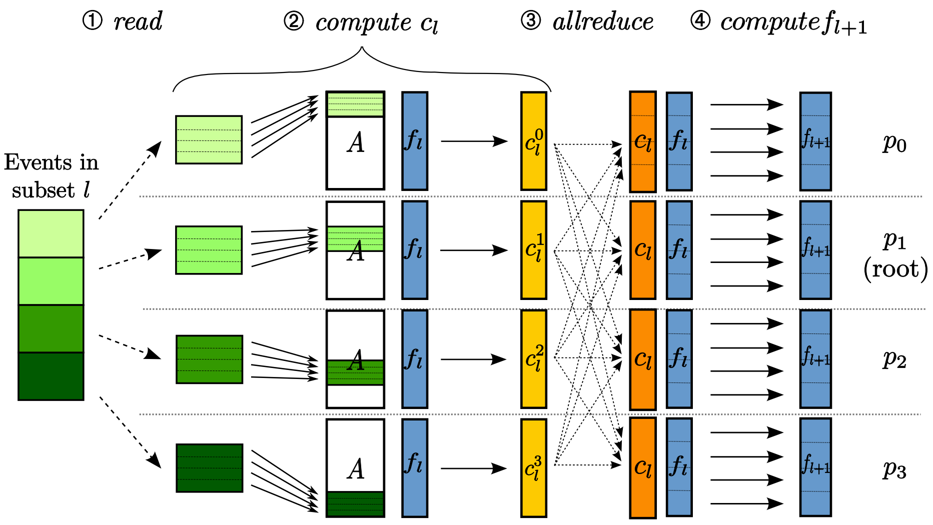


Fig. 2. The PSD strategy on a hybrid machine with 4 nodes with 4 processing units each

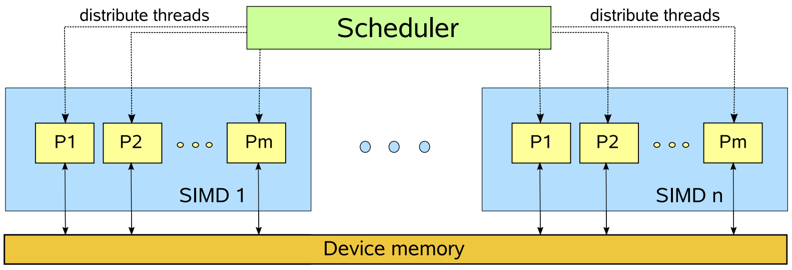


Fig. 3. Architecture of an NVIDIA GPU with n multiprocessors and m shader units

3.3 Parallelization on Graphics Processors

GPU Architecture and CUDA. Modern GPUs (Graphics Processing Units) can be used as mathematical coprocessors: they add computing power to the CPU. A GPU is a parallel machine (see Fig. 3) that consists of SIMD (Single Instruction Multiple Data) multiprocessors (ranging from 1 to 32). The stream processors of a SIMD multiprocessor are called shader units. The GPU (also called *device*) has its own fast memory with an amount of up to 4 GB. On the main board, one to four GPUs can be installed and used as coprocessors simultaneously. The GeForce 8800 GTX by NVIDIA, which we use in our experiments, provides 768 MB device memory and has 16 multiprocessors each with 8 shader units.

With CUDA (Compute Unified Device Architecture) [11], the GPU vendor NVIDIA provides a programming interface that introduces the thread-programming concept for GPUs to the C programming language. A block of threads executing the same code fragment, the so-called *kernel* program, runs on one multiprocessor. Each thread of this block runs on one of the shader units of the GPU, each unit executing the kernel on a different data element. All blocks of threads of one application are distributed among the multiprocessors by the *scheduler*. The GPU's device memory is shared among all threads.

List-Mode OSEM CUDA Implementation. The calculations for one subset on the GPU proceed as follows:

1. The CPU reads the subsets' events and copies them to the GPU device memory.
2. Each thread computes a partial sum of c_l and adds it directly to the device memory. The amount of events per thread is chosen according to the following considerations: Firstly, as many threads as possible should be started in order to hide memory latency efficiently [12]. However, each thread needs to save partial results in the device memory, which requires too much memory if one thread is started per event. Therefore, the maximum number of threads is started so that all partial results still fit into the device memory.
3. Each thread computes one voxel value for $f_{l+1} = f_l c_l$.
4. f_{l+1} is copied back to the CPU.

Note that during the computation of c_l (step 2), the threads write, as in the shared-memory implementation, directly to the shared vector c_l. In order to avoid race conditions, we again have to protect c_l with a mutex. Since this is not directly possible with CUDA (necessary mechanisms are lacking, only atomic `integer` operations exist), we decided to allow race conditions in the GPU implementation due to the following considerations:

– When two threads add one float concurrently to one voxel, then, in the worst case, one thread overwrites the other, i.e., the result will be slightly underestimated.
– The image size (e.g., $150 \cdot 150 \cdot 280 = 6.300.000$ in our experiments) is large compared to the number of parallel writing threads (e.g., 128 in our experiments); therefore, the number of race conditions and thus incorrect voxels is relatively small. We estimated experimentally that only for about 0.04 % of all writes to c_l a race condition occurs.
– Most importantly: the maximum relative error (arising from race conditions and loss of precision due to single-precision floating point values) over all voxels is less than 1 %, which leads to no visual effect on the reconstructed images.
– The goal of the majority of all mouse and rat scans in the nuclear medicine clinic is to decide whether any uptake of the radioactive substance has happened in a specific organ (e.g., the liver). For these experiments, only the high accuracy of the reconstructed image, which we preserve when allowing race-conditions, is important, and not the exact quantitative results,

which we slightly underestimate by allowing race conditions. For quantitative experiments, we can use a thread-safe shared-memory or Cell processor reconstruction.

When we use two GPUs at the same time, we have two separate device memories. The computations proceed as above, with each GPU computing half of the events during the forward-projections (step 2) and half of the sub-images during the computation of f_{l+1} (step 3). After all forward-projections, the two c_ls residing on the device memories need to be summed up.

3.4 Parallelization on the Cell Processor

Cell Architecture and Its Programming. The Cell Broadband Engine is a multiprocessor developed jointly by Sony Computer Entertainment Inc., Toshiba Corp. and IBM Corp. It consists of one PowerPC Processor Element (*PPE*) and eight processing cores called Synergistic Processor Elements (*SPEs*) (see Fig. 4). Communication is performed through the Element Interconnection Bus (*EIB*) which includes: 1) communication between PPE and SPEs, 2) access to shared memory (*main storage*) and 3) I/O. The PPE acts as controller for the SPEs by distributing computational workload and handling operating system tasks. The PPE consists of the PowerPC Processor Unit (*PPU*) and a cache. The SPEs are typically assigned to handle the computational workload of a program. Each SPE consists of a Synergistic Processor Unit (*SPU*) running at 3.2 GHz and a local store (*LS*) of 256 KB, from and to which it can transfer data from the main storage via DMA transfers. DMA transfer size ranges from 128 bytes to 16 KB. The SPUs have a local store data access rate of 51 GB/sec.

To program applications for the Cell processor, IBM provides a Software Development Kit (SDK) [13] which contains the GCC C/C++-language compilers

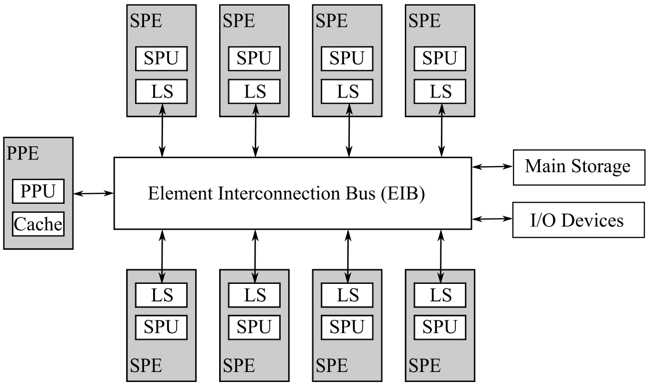


Fig. 4. Cell architecture: One PPE and eight SPEs share access on Main Storage and I/O Devices through the EIB

for the PPU and the SPU. Furthermore, the SDK includes a set of Cell processor C/C++ libraries which provide an application programming interface (API) for accessing the SPEs on PPE side. On SPE side, libraries provide DMA commands for transferring data from main storage to local store and vice versa. This includes atomic commands which provide mutual exclusion to avoid race conditions.

List-Mode OSEM Cell Implementation. The calculation of one subiteration of our example algorithm on p SPEs proceeds as follows:

1. The PPE reads the subsets' events and stores them in the main storage. Afterwards, the PPE sends each SPE a message to start computations.
2. Each thread computes a partial sum of c_l and adds it directly to the device memory. Since all threads write simultaneously to the shared c_l, we use an atomic operation.
3. The reconstruction image is divided into sub-images f^j. Each SPE computes $f_{l+1}^j = f_l^j\, c_l^j$ on its sub-image.

Note that for the forward projection (step 2), the programmer has to organize transferring the required voxels of f_l and c_l from main storage to the SPEs' local store. Since the minimum DMA transfer size is 128 bytes, then 128 bytes instead of 4 bytes for one `float` have to be transferred for each voxel of f_l, when computing $c_{l,j}$ and $c_l + c_{l,j}$. Since a path, in almost all cases, crosses through several y- and z-planes of the 3D image, the bulk of additional transferred voxels of f_l cannot be used in the following computations (see Fig. 5). Using the minimum transfer size of 128 bytes, an average of 1.6 of the 32 transferred voxels, i.e., 5 % of each DMA transfer are used.

When using two Cell processors, i.e., two PPEs and altogether sixteen SPEs, we have two main storages, such that each PPE only communicates with its

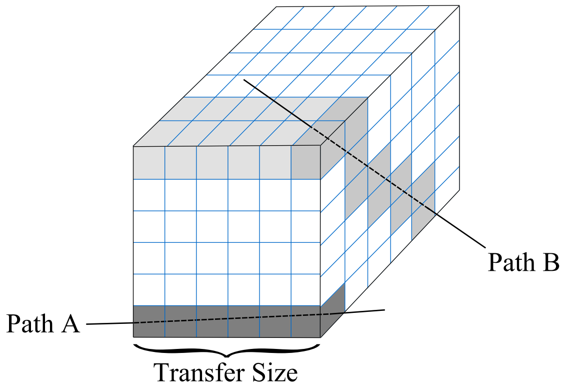


Fig. 5. Two example paths: Path A shows a best-case scenario as only one DMA transfer is required. Path B shows a worst-case scenario as ten DMA transfers are required.

SPEs. The communication between both main storages and the SPE management is transparent to the programmer and thus the programmer can develop his code as if there were only one PPE with sixteen SPEs.

4 Runtime Experiments and Architecture Comparison

Since the image size of the list-mode OSEM on all four architectures presented in the previous section has only little influence on scalability [7,8], we restrict our considerations to the typical image size of $N = (150 \times 150 \times 280)$. We use 10^7 events in 10 subsets acquired during a 15 minute mouse scan of the quadHIDAC [1] small-animal PET scanner.

We use the following parallel machines in our experiments:

Quad-core Processor: Intel Core 2 Quad processor with four cores running at 2.83 GHz. Two cores share 6 MB level 2 cache and all cores share the 4 GB main memory. The memory throughput is up to 11 GB/s.

Hybrid Cluster: Arminius cluster with 200 Dual INTEL Xeon 3.2 GHZ 64bit nodes, each with 4 GByte main memory, connected by an InfiniBand network. To exploit the fast InfiniBand interconnect (point-to-point throughput of up to 900 MB/s), we used the Scali MPI Connect [14] implementation on this machine.

GPUs: Two GPUs of the type NVIDIA GeForce 8800 GTX. They have 16 SIMD-multiprocessors, each with 8 shader units running at 1.35 GHz. The device memory is 768 MB. The measured throughput between device and CPU main memory is 1.5 GB/s. The multi-processor to device throughput is 86 GB/s.

Cell Processor: A QS21 Blade Center equipped with two Cell processors. Each Cell processor consists of one PPE running at 3.2 GHz with 512 KB L2 cache and 1 GB main memory and 8 SPEs running also at 3.2 GHz equipped with 256 KB local storage. The EIB supports a peak bandwidth of 204.8 GB/s and the integrated memory controller (MIC) provides a peak bandwidth of 25.6 GB/s to the DDR2 memory.

4.1 Performance

In the following, we analyze the performance of the parallel implementation in terms of total runtime (Fig. 6) and scalability (Fig. 7).

The hybrid cluster outperforms all other architectures with a minimum reconstruction time of ≈ 15 seconds on 64 processors. However, the implementation does not scale well: the speedup on 16 processors is ≈ 7 and thus less than 50% of the ideal speedup. Moreover, runtime deteriorates for 128 processors. For each subiteration, one `MPI_Allreduce` is performed which includes at least $log_2(p)$ communication rounds. For example, on 64 processors, at least 8 communication rounds that each require a 48 MB image to be sent from one process to another are performed. Hence, with a point-to-point bandwidth of 900 MB/s,

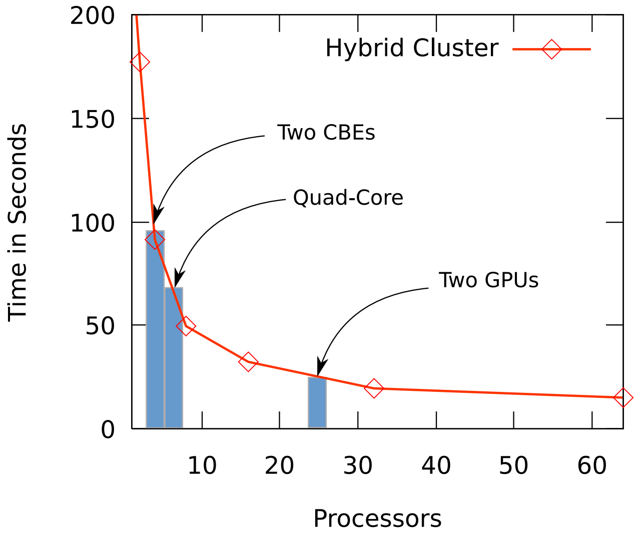


Fig. 6. Runtime comparison: hybrid cluster (curve); quad-core processor, two GPUs and two Cell processors (bars)

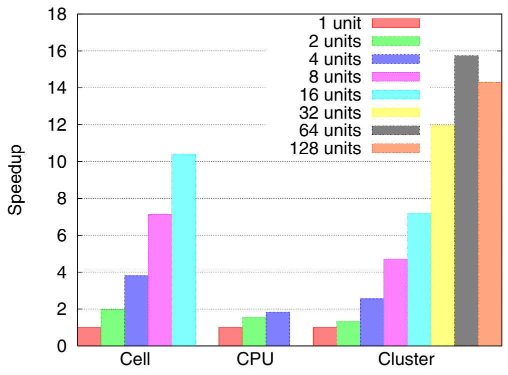


Fig. 7. Scalability comparison: hybrid cluster (units $\widehat{=}$ processors), quad-core processor (units $\widehat{=}$ cores) and two Cell processors (units $\widehat{=}$ SPEs)

in the best case, 0.4 seconds (of the overall 1.5 seconds computation time) are spent in MPI communication for each subset. Thus, the application scales rather poorly (refer to [5] for more details).

The two GPUs are only 1.6 times slower than the 64 processor cluster. However, runtime only decreased from 33 seconds to 24 seconds when going from one to two GPUs. Since this is less than 50 %, we can only expect little speedup by adding more GPUs over the main board's PCI Express slots. Furthermore, since only an insufficient profiling tool exists, it is quite difficult to assess what the current performance bottleneck is. Therefore, we cannot determine if more shader units or increased memory bandwidth would speed up our application.

The quad-core processor is ≈ 5 times slower than the 64 processors of the hybrid cluster. The main limiting scalability factor on the quad-core processor is, as on the cluster, the restricted memory bandwidth.

Although four times as many cores are available in two Cell processors with overall 16 SPEs as on the quad-core processor, two Cells still provide the worst runtime. As described in Section 3.4, only 5 % of each 128 byte DMA transfer is actually used in computations. Therefore, a lot of time is spent in transferring large amounts of unused data. Hence, the minimum size of 128 bytes per DMA transfer is an important limiting factor of the Cell architecture in our application.

4.2 Programming Comfort

Today, all four programming tools we use to implement the parallel algorithm, can be seen as the standard to program the according architectures. Therefore, comparing the programming tools allows us to compare the architectures with respect to their programmability.

Implementing our application for shared-memory machines with OpenMP is obviously the easiest choice: for our parallel implementation, we only added three lines of code, one for each loop and the critical section. In [15] we have shown that the recently introduced Intel's Threading Building Blocks (TBB) library provides better performance for our application; however, using TBB required a complete rewrite of our code because C++ constructs are required. But once rewritten, TBB not only provides better performance, but, additionally, task-parallel concepts can be easily used, for example to overlap the reading of data with computations.

For the MPI implementation, the communication with `MPI_Allreduce` had to be introduced to the sequential implementation. While this work was straightforward, it was more difficult to efficiently implement the processe's concurrent access to the remote file system when reading the input events. Nonetheless, programming with MPI for our application was quite uncomplicated, when compared to the GPU and Cell programming. Since the OpenMP implementation is already available, no extra work needed to be done in order to implement the algorithm on the hybrid cluster.

Programming with CUDA has some significant programming difficulties compared to the other three architectures:

– Debugging and profiling is quite difficult on the GPU, especially because the SUs cannot perform I/O operations and therefore not even a `printf` is possible.
– Multiple GPUs have to be managed explicitly by the programmer.
– There exist no mutexes, semaphores or atomic floating point operations in CUDA; therefore, we can only use the GPU implementation, as described in Section 3.3 for non-quantitative reconstructions where race conditions can be accepted.

– CUDA is only available on NVIDIA GPUs. OpenCL, a novel programming standard that closely resembles NVIDIA's CUDA, has recently been specified [18]. OpenCL will be supported, amongst others, by AMD and NVIDIA GPUs.

For our application, we determined the following main disadvantages of the Cell processor and the Cell SDK compared to the three other programming environments:

– The restricted DMA transfer sizes (both in hardware and software) on the Cell processor entail two disadvantages: 1) Programming is more difficult, because often data elements have to be aligned to DMA transfer sizes manually by the programmer, and 2) the minimum size of 128 Bytes limits the performance of our parallel Cell processor implementation enormously.
– In order to hide memory latency on the Cell processor, the programmer has to explicitly implement double-buffering schemes that allow loading data to local storages during simultaneous computations. On the other architectures, memory latency is either hidden by switching among threads (GPU) or by a complex, hardware-managed cache hierarchy; thus, programming on the Cell processor is significantly more complicated.

Summarizing, we can say that the Cell SDK provides the lowest abstraction level and, therefore, the Cell processor is the most difficult to program. While CUDA provides a higher abstraction level, its lack of sufficient debugging tools makes programming GPUs with this framework more tedious than programming with MPI. Finally, OpenMP provides the highest abstraction level and is the easiest to use.

4.3 Accessibility

Multi-core computers, workstations with CUDA-enabled GPUs and a Cell blade can be run in a local network. Therefore, accessibility is high for all three. This is especially true for multi-core processors, because they are available in virtually every off-the shelf computer today. On the contrary, buying a cluster for a medical clinic will most likely be too expensive. Furthermore, accessing a remote cluster results in two problems: 1) additional administrative effort is necessary in order to reconstruct images on a remote cluster, e.g., the locating of free resources, and 2) in- and output data have to be transferred over the Internet from and to the cluster. While the first problem can be solved with the grid system we introduce in Section 5, the second problem leads to considerably longer runtime. For example, sending a 100 MB input dataset with 10^7 events from the University of Münster to the above mentioned Arminius cluster at the University of Paderborn (distance $\approx$ 100 km) takes $\approx$ 15 s. Hence, runtime for the reconstruction of 10^7 events on 64 processors of the hybrid cluster increases from about 15 seconds (Table 1) to about 30 seconds and is thus 1.25 times slower than the GPU reconstruction.

Table 1. Average measured runtime of the list-mode OSEM algorithm for 10^7 events in 10 subiterations and estimated price for the corresponding architecture

Architecture:	Multi-core 4 cores	Hybrid 64 processors	GPU 2 devices	Cell 2 cell procs
Runtime:	72.6 s	14.8 s	24.4 s	99.8 s
Est. Price:	€1.500	€1.500.000	€2.000	€5.500

4.4 Cost-Effectiveness

The Cell processor demonstrated rather poor performance for our algorithm and is not cheaper than GPUs and multi-core processors; thus, it is less cost-effective. We estimate that a workstation equipped with a high-end CPU and a low-cost GPU to be about as expensive as a workstation with a medium-cost CPU and two high-end GPUs. But since the GPU outperforms the multi-core CPU by a factor of two, the GPU is more cost-effective.

Buying and maintaining a cluster is quite expensive (about 1.5 million euros for the cluster used in our experiments). Therefore, a cluster is definitely less cost-effective than the other options. A second possibility is to buy computing hours on a cluster. However, monthly costs for reliable cluster computation time are quite high (e.g., 900 euros for 4.000 processor hours [16] which corresponds to 20 reconstructions per day on 64 processors). Thus a GPU would be paid off after three months and thus again provides significantly better cost-effectiveness. Summarizing, GPU proves to be the most cost-effective parallel architecture, followed by the multi-core CPU.

In the next section, we show how our grid system chooses the most suitable architecture for a given reconstruction task from the available parallel machines.

5 Grid System for PET Reconstruction

MIRGrid (Medical Image Reconstruction Grid) [17] is an experimental grid system that we have developed to integrate in a single application all steps of the imaging process, which are traditionally performed by the user using different software tools: from reading the raw data acquired by the scanner, over transparent parallel reconstruction to the visualization and storage of reconstructed images.

Additionally to the standard 3D list-mode OSEM reconstruction, the system seamlessly integrates *dynamic* and *gated* reconstructions. In dynamic studies, a 4D image sequence is generated that captures the radioactive substance's distribution over time. The grid system allows the medical user to divide the complete list-mode dataset into time intervals. The corresponding 3D reconstructions are then performed transparently for the user. In gated reconstructions, one heart or respiratory cycle is divided into a number of gates and each gate is reconstructed independently by the grid system.

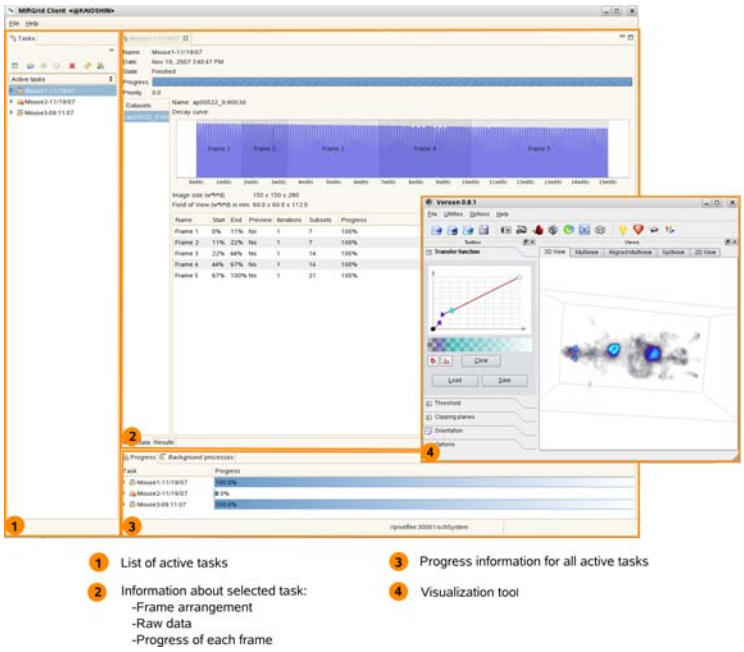


Fig. 8. A screenshot of the client's main window in MIRGrid

The MIRGrid system is composed of three modules: the client (see Fig. 8), the scheduler and the runtime system. When a user starts the client program on his local desktop computer, the client connects over the Internet to the scheduler-server (the machine on which scheduler and runtime system run).

After the user has chosen the raw data previously collected by the scanner and the parameters for reconstruction, the client sends the data and the parameters to the scheduler.

Transparently to the user, the scheduler then assigns the reconstruction to a HPC, and the runtime system starts and monitors the reconstruction on that HPC. When the reconstruction is finished, the result images are sent back to the client where they are visualized and stored.

The grid system currently supports shared-memory machines and cluster computers. The system is installed at the nuclear medicine clinic in Münster and is currently tested before going into productive use in a few months. We plan to integrate support for GPUs on MIRGrid in the near future.

6 Conclusion

Our comparison of different parallelization approaches for an important medical imaging application has brought several important findings.

The GPU proved to be the most cost-effective architecture. Since it is also quite well accessible, it is suitable for standard image reconstruction tasks. However, if very accurate quantitative reconstruction results are required, then multi-core processors or hybrid clusters are to be preferred, because, in contrast to the GPU, they allow to prevent race conditions. Also, programming for the GPU is quite tedious and error-prone; therefore, for research code that is continuously enhanced and tested, multi-core processors with OpenMP are to be preferred.

Currently, new algorithms are being developed in our collaborative research group that are even more compute-intensive than the standard list-mode OSEM. For example, we estimate that the so called *EM-TV* algorithm [19] applied to 3D PET and an advanced scatter correction method [20] will both be one to two orders of magnitude more compute-intensive than the current algorithm. In order to use such algorithms, clusters of multi-core processors will probably be the architecture to target.

Our analysis and experiments demonstrated that the Cell processor is not very useful for the list-mode OSEM reconstruction, because it provides poor performance and is quite difficult to program.

Finally, our MIRGrid system transparently chooses the most suitable architecture for a given reconstruction task from the set of available parallel machines.

Acknowledgments

We thank the NVIDIA corporation for the donation of graphics hardware used in our experiments. We thank IBM Deutschland for letting us access their QS21 Cell blades. This work was partly funded by the Deutsche Forschungsgemeinschaft, SFB 656 MoBil (Projects B2, B3, PM6).

References

1. Schäfers, K.P., Reader, A.J., Kriens, M., Knoess, C., Schober, O., Schäfers, M.: Performance Evaluation of the 32-Module QuadHIDAC Small-Animal PET Scanner. Journal Nucl. Med. 46(6), 996–1004 (2005)
2. Reader, A.J., Erlandsson, K., Flower, M.A., Ott, R.J.: Fast Accurate Iterative Reconstruction for Low-Statistics Positron Volume Imaging. Phys. Med. Biol. 43(4), 823–834 (1998)
3. Shepp, L.A., Vardi, Y.: Maximum Likelihood Reconstruction for Emission Tomography. IEEE Trans. Med. Imag 1, 113–122 (1982)
4. Siddon, R.L.: Fast Calculation of the Exact Radiological Path for a Three-Dimensional CT Array. Medical Physics 12(2), 252–255 (1985)
5. Hoefler, T., Schellmann, M., Gorlatch, S., Lumsdaine, A.: Communication Optimization for Medical Image Reconstruction Algorithms. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 75–83. Springer, Heidelberg (2008)
6. Schellmann, M., Gorlatch, S.: Comparison of Two Decomposition Strategies for Parallelizing the 3D List-Mode OSEM Algorithm. In: Proceedings Fully 3D Meeting and HPIR Workshop, pp. 37–40 (2007)

7. Schellmann, M., Vörding, J., Gorlatch, S., Meiländer, D.: Cost-Effective Medical Image Reconstruction: From Clusters to Graphics Processing Units. In: Proceedings of the 2008 Conference on Computing frontiers, pp. 283–292. ACM, New York (2008)
8. Meiländer, D., Schellmann, M., Gorlatch, S.: Implementing a Data-Parallel Application with Low Data Locality on Multicore Processors. In: International Conference on Architecture of Computing Systems - Workshop Proceedings, pp. 57–64. VDE (2009)
9. OpenMP Architecture Review Board. OpenMP Application Program Interface (May 2008)
10. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, http://www.mpi-forum.org
11. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture, http://developer.nvidia.com/object/cuda.html
12. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Hwu, W.: Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In: PPoPP 2008: Proc. of the 13th ACM SIGPLAN Symposium, pp. 73–82 (2008)
13. IBM. Software Development Kit for Multicore Acceleration Version 3.0, http://www.ibm.com/developerworks/power/cell/
14. Scali MPI connect, http://www.scali.com/
15. Kegel, P., Schellmann, M., Gorlatch, S.: Using OpenMP and Threading Building Blocks for Parallelizing Medical Imaging: A Comparison. In: Euro-Par 2009 - Parallel Processing. LNCS, vol. 5704. Springer, Heidelberg (to appear, 2009)
16. Tsunamic Technologies Inc., Cluster computing on demand, http://www.clusterondemand.com/
17. Schellmann, M., Böhm, D., Wichmann, S., Gorlatch, S.: Towards a Grid System for Medical Image Reconstruction, pp. 3019–3025. IEEE Computer Society Press, Los Alamitos (2007)
18. Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/
19. Brune, C., Sawatzky, A., Burger, M.: Bregman-EM-TV Methods with Application to Optical Nanoscopy. In: Proceedings of the 2nd International Conference on Scale Space and Variational Methods in Computer Vision. LNCS, vol. 5567, pp. 235–246. Springer, Heidelberg (2009)
20. Kösters, T., Wübbeling, F., Natterer, F.: Scatter Correction in PET Using the Transport Equation. In: IEEE Nuclear Science Symposium and Medical Imaging Conference Record, pp. 3305–3309. IEEE, Los Alamitos (2006)

Author Index