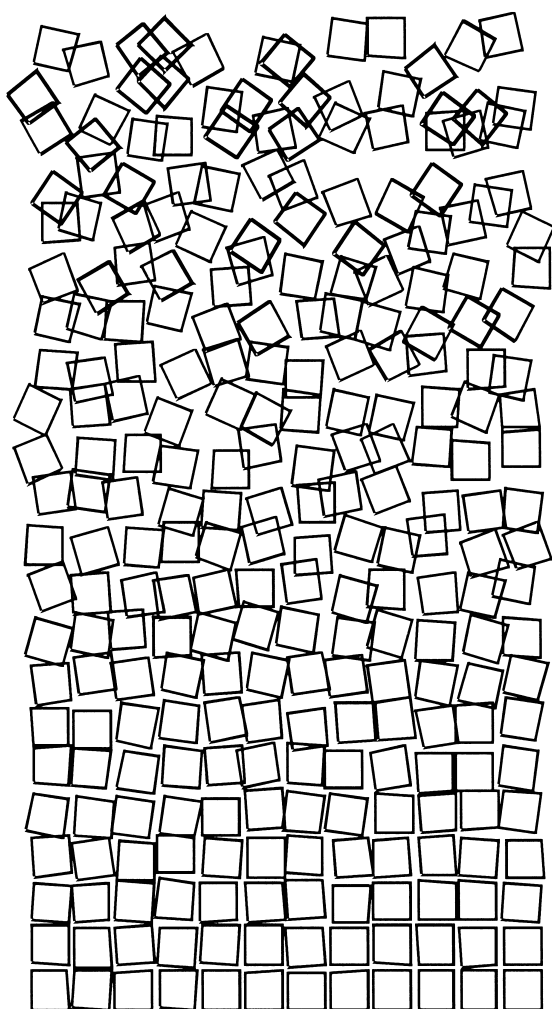


Inhalt



Computergraphik von: Georg Nees, Generative Computergraphik

22. PARS-Workshop Parsberg (Full Papers)	5
22. PARS-Workshop Parsberg (Poster)	123
PARS (Berichte, Aktivitäten, Satzung)	149
<u>23. PARS-Workshop 2010</u> (Hannover, 23. Februar 2010) http://pv.fernuni-hagen.de/pars/2010/	157
<u>ARCS 2010</u> (Hannover, 22. – 25. Februar 2010)	158
<u>Euro-Par 2010</u> (Ischia-Naples, Italien, 31. August – 3. September 2010)	159

Aktuelle PARS-Aktivitäten unter:

- <http://www.fg-pars.gi-ev.de/>

PARS-Mitteilungen

Gesellschaft für Informatik e.V., Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware

Offizielle bibliographische Bezeichnung bei Zitaten:

Mitteilungen - Gesellschaft für Informatik e. V.,

Parallel-Algorithmen und Rechnerstrukturen, ISSN 0177 - 0454

PARS-Leitungsgremium:

Prof. Dr. Helmar Burkhart, Univ. Basel
Dr. Andreas Döring, IBM Zürich
Prof. Dr. Dietmar Fey, Univ. Erlangen
Prof. Dr. Rolf Hoffmann, Sprecher, TU Darmstadt
Prof. Dr. Wolfgang Karl, Univ. Karlsruhe
Prof. Dr. Jörg Keller, FernUniversität Hagen
Prof. Dr. Christian Lengauer, Univ. Passau
Prof. Dr.-Ing. Erik Maehle, Universität zu Lübeck
Prof. Dr. Ernst W. Mayr, TU München
Prof. Dr. Wolfgang E. Nagel, TU Dresden
Dr. Karl Dieter Reinartz, stellv. Sprecher, Univ. Erlangen-Nürnberg
Prof. Dr. Hartmut Schmeck, Univ. Karlsruhe
Prof. Dr. Theo Ungerer, Univ. Augsburg
Prof. Dr. Helmut Weberpals, TU Hamburg Harburg

Die PARS-Mitteilungen erscheinen in der Regel einmal pro Jahr. Sie befassen sich mit allen Aspekten paralleler Algorithmen und deren Implementierung auf Rechenanlagen in Hard- und Software.

Die Beiträge werden nicht redigiert, sie stellen die Meinung des Autors dar. Ihr Erscheinen in diesen Mitteilungen bedeutet keine Einschränkung anderweitiger Publikation.

Die Homepage

<http://www.fg-pars.gi-ev.de/>

vermittelt aktuelle Informationen über PARS.

22. PARS – Workshop

veranstaltet am 4./5. Juni 2009 in
Parsberg in der Oberpfalz

Ziel des PARS-Workshops war die Vorstellung wesentlicher Aktivitäten im Arbeitsbereich von PARS und ein damit verbundener Gedankenaustausch. Mögliche Themenbereiche waren etwa:

- **Parallele Algorithmen (Beschreibung, Komplexität, Anwendungen)**
- **Parallele Rechenmodelle und parallele Architekturen**
- **Parallele Programmiersprachen und Bibliotheken**
- **Parallele Programmierung und Programmparallelisierung**
- **Software Engineering für parallele und verteilte Systeme**
- **Vernetzte Systeme und Anwendungen, Grid Computing, Cloud Computing**
- **Verbindungsstrukturen und Hardwareaspekte (z. B. rekonfigurierbare Systeme)**
- **Neue Technologien und Architekturen (SoC, Multicores, PIM, STM etc.)**
- **Alternative Technologien (Quantencomputing, DNA-Computing)**
- **Parallel Organic Computing**
- **Parallelverarbeitung im Unterricht (Erfahrungen, E-Learning)**
- **Parallele Methoden und verteiltes Rechnen in der Bioinformatik**

Programmkomitee: *H. Burkhart, Basel • A. Döring, Zürich • D. Fey, Jena • R. Hoffmann, Darmstadt • F. Hoßfeld, Jülich • W. Karl, Karlsruhe • J. Keller, Hagen • Chr. Lengauer, Passau • E. Maehle, Lübeck • E. W. Mayr, München • F. Meyer auf der Heide, Paderborn • W. E. Nagel, Dresden • K. D. Reinartz, Höchstadt • H. Schmeck, Karlsruhe • P. Sobe, Lübeck • T. Ungerer, Augsburg • H. Weberpals, Hamburg Harburg*

Veranstalter: GI/ITG-Fachgruppe PARS, <http://www.fg-pars.gi-ev.de/>

Organisation: Prof. Dr. Wolfgang Karl, Institut für Technische Informatik
Universität Karlsruhe (TH), Kaiserstraße 12, D-76128 Karlsruhe
Tel.: 0721-608-3771, Fax: 0721-608-3962
E-Mail: karl@ira.uka.de

Prof. Dr.-Ing. Rolf Hoffmann (PARS-Sprecher), FG Rechnerarchitektur,
FB Informatik, TU Darmstadt, Hochschulstr. 10, D-64289 Darmstadt
Tel.: 06151-16-3611/3606, Fax: 06151-165410
E-Mail: hoffmann@informatik.tu-darmstadt.de

**Technische
Organisation:**

Dr.-Ing. Wolfgang Heenes, FG Rechnerarchitektur,
FB Informatik, TU Darmstadt, Hochschulstr. 10, D-64289 Darmstadt
Tel.: 06151-165312, Fax: 06151-165410
E-Mail: heenes@ra.informatik.tu-darmstadt.de

Parallele Algorithmen und Anwendungen 1

A Parallel PDE-Constrained Optimization Framework for Biomedical Hyperthermia Treatment Planning	7
---	----------

Madan Sathe, Olaf Schenk, Matthias Christen, Helmar Burkhart

Mesosopic Simulations of Hydrodynamic Interactions Using a Particle-Based Model on Cell Broadband Engine	17
---	-----------

Annika Schiller, Godehard Sutmann

HICFD - Hocheffiziente Implementierung von CFD-Codes für HPC-Many-Core-Architekturen	27
---	-----------

T. Gerhold, J. Jägersküpper, N. Kroll, E. Kügeler, C.Voigt, A. Basermann, H.-P. Kersken, A. Schreiber, T. Alrutz, C. Simmendinger, K. Feldhoff, R. Müller-Pfefferkorn, P. Aumann, O. Knobloch, M. Puetz, J. Hunger, C. Zscherp

Parallele Rechenmodelle und Architekturen

Global Cellular Automata: a Path from Parallel Random Access Machines To Practical Implementations	36
---	-----------

Jörg Keller, Andre Osterloh

Network Optimization of a Multiprocessor Architecture for the Massively Parallel Model GCA	48
---	-----------

Christian Schäck, Wolfgang Heenes, Rolf Hoffmann

Architecture and Implementation of an MMU on a Server-class Infiniband HCA	58
---	-----------

Andreas Döring

Grid Computing

Grid Workflow Recovery as Dynamic Constraint Satisfaction Problem	67
--	-----------

Stanimir Dragiev, Jörg Schneider

A Dependable Middleware for Enhancing the Fault Tolerance of Distributed Computations in Grid Environments	74
---	-----------

Andre Luckow

Algorithmen und Anwendungen 2

Efficient Simulation of Fork Programs on Multicore Machines	84
<i>Jörg Keller, Christoph Kessler, Bert Wesarg</i>	

Putting 'p' in RabbitCT - Fast CT Reconstruction Using a Standardized Benchmark	91
<i>Hannes Hofmann, Benjamin Keck, Christopher Rohkohl, Joachim Hornegger</i>	

Parallele Programmierung und Virtualisierung

TBCA - Eine Thread-basierte Cluster-Architektur	101
<i>Martin Meierhöfer, Bernd Klauer</i>	

On the performance of virtualized cluster computing	106
<i>Stephan Briskorn, Udo Hönig und Wolfram Schiffmann</i>	

Providing Guidance Information for Application-Mapping on Heterogeneous Parallel Systems	115
<i>Fabian Nowak, Rainer Buchty</i>	

A Parallel PDE-Constrained Optimization Framework for Biomedical Hyperthermia Treatment Planning

Madan Sathe, Olaf Schenk, Matthias Christen, and Helmar Burkhart

Madan Sathe, Olaf Schenk, Matthias Christen, Helmar Burkhart
Computer Science Department, University of Basel, Klingelbergstrasse 50, CH-4056 Basel
{madan.sathe, olaf.schenk, m.christen, helmar.burkhart}@unibas.ch

Abstract. We present a PDE-constrained optimization algorithm which is designed for parallel scalability on distributed-memory architectures with thousands of cores. The method is based on a line-search interior-point algorithm for large-scale continuous optimization, it is matrix-free in that it does not require the factorization of derivative matrices. Instead, it uses a new parallel and robust iterative linear solver on distributed-memory architectures. We will show almost linear parallel scalability results with 256 cores for the optimization problem, which is an emerging biomedical application and is related to antenna identification in hyperthermia cancer treatment planning. Additionally, we will discuss promising existing parallel matching algorithms which could further enhance the parallel scalability of general nonlinear optimization solvers.

Key words: PDE-constrained optimization, large-scale parallel optimization, biomedical application, sparse linear solver, parallel weighted matching algorithm

1 Introduction

All signs in the world of high-performance computing point towards a future that is populated with multi- and many-core processing technologies. The emergence of multi-core processors such as GPUs and Cell Broadband Engines and the appearing of high-performance cluster with thousands of cores allow solving highly complex computational problems arising in emerging applications such as in biomedical or fluid dynamical environments.

In many application areas there is a great interest in solving nonlinear optimization problems of extremely large sizes. For example, if the constraints of the problem correspond to discretized partial-differential equations, then the accuracy of a solution with respect to this infinite-dimensional problem is directly related to the size of the largest discrete approximate problem that can be solved. The storage and factorization of explicit derivative matrices is intractable, so researchers and practitioners are often forced to seek alternatives to available optimization techniques.

Very often, these computational applications are constructed with software toolkits with flexible and easy-to-use interfaces. These software tools can be put together in order to deal with large-scale parallel applications. The complete optimization process could be constructed, e.g., from an adaptive meshing toolkit, a dynamic load-balancing toolkit, a nonlinear optimization toolkit (here IPOPT), a parallel linear solver (here PSPIKE) and some parallel visualization tools.

In this paper, we will use an inexact primal-dual barrier interior-point optimizer IPOPT and add a new scalable and robust solver — PSPIKE. It will be used to solve a large-scale biomedical PDE-constrained optimization application in parallel on distributed-memory architectures. The underlying framework of the new solver PSPIKE is based on the well-known shared-memory parallel sparse direct solver PARDISO [10], which typically scales up to sixteen cores. In order to address scalability up to thousands of cores, the new solver PSPIKE has been very recently developed and represents a significant extension to existing parallel methods. In this paper, we will show almost linear scalability results for the optimization application up to 256 cores. Additionally, we will discuss parallel algorithms for a combinatorial problem, a matching problem, which can further enhance the parallel scalability of current linear solvers.

2 PDE-Constrained Biomedical Hyperthermia Cancer Treatment Planning

Biomedical hyperthermia cancer treatment is a promising therapeutical option in oncology [7]. Heating tumors above a temperature of about 45°C results in preferential killing of tumor cells and makes them more susceptible to an accompanying radio or chemo therapy (see Figure 1). Modern hyperthermia applicators operating at around 100 MHz provide a larger number of antennas for which the amplitude and phase can be controlled independently which permits shifting the focus and preventing hotspots. The optimal hyperthermia treatment planning can be formulated as a nonlinear optimization problem, in which the Pennes Bio-heat equations [8] appear as important constraints — a mathematical task known as *PDE-constrained optimization*. Solving the PDE constrained optimization problem presents a frontier problem in scientific computing. The size, complexity and infinite-dimensional nature of PDE-constrained optimization problems present significant challenges for general-purpose optimization algorithms and, typically, Tikhonov regularization, iterative solvers, preconditioning, inexactness and parallel implementations are necessary to cope with the numerical challenges.

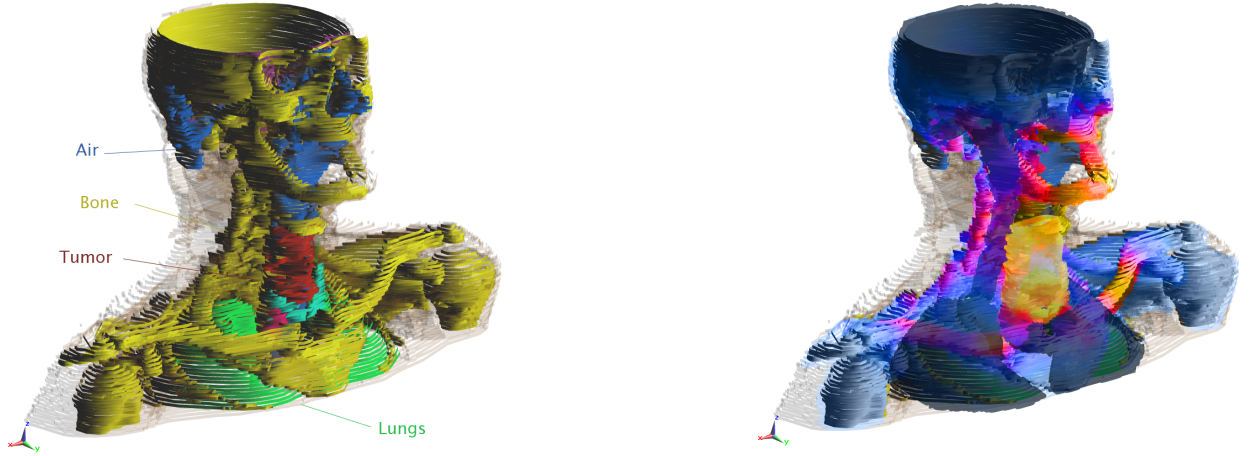


Fig. 1. Patient model and simulated energy absorption temperature distribution. The images show a temperature distribution within an iteration of our parallel PDE-constrained optimization process.

For designing an individually optimal therapy, amplitudes and phases of the antennas have to be selected such that the tumor temperature is maximized up to a target therapeutical temperature $\mathbf{T}_{ther}$ of 43°C. In order not to damage healthy tissue, certain temperature constraints have to be respected. The induced temperature distribution is essentially described by the elliptic bio heat transfer equation [8]. The aim is to predict the temperature distribution $\mathbf{T}$ (*state variables*) which depends on the complex control of the antennas $\mathbf{u}_i$ and E_i (*control variables*). This leads to the following large-scale nonlinear optimal PDE control problem and the goal is to minimize the objective function [7]:

$$F = \int_{x \in \Omega_t} (\mathbf{T}_{ther} - \mathbf{T})^2 d\Omega + \int_{x \notin \Omega_t, \mathbf{T} > \mathbf{T}_{health}} (\mathbf{T} - \mathbf{T}_{health})^2 d\Omega \quad (1a)$$

subject to

$$-\nabla \cdot (k \nabla \mathbf{T}) + \rho_b c_b \rho \omega (\mathbf{T} - \mathbf{T}_b) = \frac{\rho \sigma}{2} \left| \sum_i u_i E_i \right|^2 \quad \text{in } \Omega \quad (1b)$$

$$k \partial_n \mathbf{T} = q_{const} \quad \text{on } \partial \Omega \quad (1c)$$

$$\mathbf{T}|_{\Omega/\Omega_t} < \mathbf{T}_{lim} \quad (1d)$$

$$\min\{|\mathbf{u}_i|\} \geq \alpha \max\{|\mathbf{u}_i|\} \quad (1e)$$

Here, k is the thermal conductivity, ρ the density (ρ_b : blood density), ω the perfusion rate, σ the electrical conductivity, $\mathbf{T}$ the temperature and $\mathbf{T}_b$ the arterial blood temperature and these tissue parameters depend also on the temperature. Ω is the part of the patient's body that is affected, $\Omega_t \subset \Omega$ is the domain occupied by tumor tissue. The complex control of antenna is defined by $\mathbf{u}_i$ and E_i , and the temperature is $\mathbf{T}_{ther} = 43^\circ\text{C}$, $\mathbf{T}_{health} = 42^\circ\text{C}$, and $\mathbf{T}_{lim} = 44^\circ\text{C}$. α depends on the HTP applicator that is used in the therapy.

In our application there is great interest in solving optimization problems of extremely large sizes. Since the constraints of the problem correspond to a discretized biomedical PDE, the accuracy of the optimization solution with respect to this infinite-dimensional problem is directly related to size of the largest discrete approximate problem that can be solved. The challenge is thus to design a constrained optimization algorithm that emulates an efficient nonlinear programming approach. Recent advances in optimization algorithms [4,5,12] combined with scalable robust appropriate preconditioners will result in PDE-constrained optimization applications that can often scale to millions of optimization variables. We note that our method has much in common with the algorithms in [2,3,6] as we follow a *full-space*, or *all-at-once*, approach for PDE-constrained problems. The major difference, however, is that we have presented in [5] conditions that guarantee the global convergence of the inexact optimization algorithm.

3 The Parallel Scalable PSPIKE Algorithm

3.1 Motivation

After applying a finite-difference discretization, the PDE-constrained optimization problem (1) can be transformed into a nonlinear programming problem (NLP) given by a nonconvex objective function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ and constraint functions $c(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, which are both assumed to be twice continuously differentiable. The objective is to find a local solution of the optimization problem.

$$\min_{x \in \mathbb{R}^n} f(x) \quad (2a)$$

$$\text{s.t. } c(x) = 0 \text{ and } x_L \leq x \leq x_U. \quad (2b)$$

Growing interest in efficient optimization methods has led to the development of interior-point or barrier methods for large-scale nonlinear programming [12]. The computational efficiency of all PDE-constrained optimization applications strongly depends on the efficiency of the numerical linear algebra kernel to solve the symmetric indefinite Karush-Kuhn-Tucker systems $Ax = b$ of optimality.

In this work, we will focus on a new method — the **PSPIKE** algorithm — which represents a highly scalable parallel solver on distributed memory architectures.

3.2 The Basic Parallel PSPIKE Algorithm

Let $Ax = f$ be a nonsymmetric diagonally dominant system of linear equations where A is of order n and bandwidth $2m+1$. Unlike classical banded solvers such as those in Lapack which are based on LU-factorization of A , the spike algorithm [1,9] is based on the factorization $A = D \times S$, in which D is a block-diagonal matrix and S is the spike matrix as shown in Figure 2 for three partitions. Note that the block diagonal matrices A_1 , A_2 , and A_3 are

$$A = D \times S$$

Fig. 2. Decomposition where $A = D * S$, $S = D^{-1}A$, $B_j, C_j \in \mathbb{R}^{m \times m}$

nonsingular by virtue of the diagonal dominance of A . For the example in Figure 2, the basic Spike algorithm consists of the following stages.

- **Stage 1:** Obtain the LU-factorization (without pivoting) of the diagonal blocks A_j (i.e. $A_j = L_j U_j, j = 1, 2, 3$)
- **Stage2:** Forming the spike matrix S and updating the right hand side
 - (i) solve $L_1 U_1 [V_1, g_1] = [(\begin{smallmatrix} 0 \\ B_1 \end{smallmatrix}), f_1]$
 - (ii) solve $L_2 U_2 [W_2, V_2, g_2] = [(\begin{smallmatrix} 0 \\ C_2 \end{smallmatrix}), (\begin{smallmatrix} 0 \\ B_2 \end{smallmatrix}), f_2]$
 - (iii) solve $L_3 U_3 [W_3, g_3] = [(\begin{smallmatrix} 0 \\ C_3 \end{smallmatrix}), f_3]$ $f_j, i \leq j \leq 3$, are the corresponding partitions of the right hand side f .
- **Stage 3:** Solving the reduced system,

$$\begin{bmatrix} I & V_1^{(b)} & 0 & 0 \\ W_2^{(t)} & I & 0 & V_2^{(t)} \\ W_2^{(b)} & 0 & I & V_2^{(b)} \\ 0 & 0 & W_3^{(t)} & I \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} g_1^{(b)} \\ g_2^{(t)} \\ g_2^{(b)} \\ g_3^{(t)} \end{bmatrix} \quad (3)$$

where $(V_i^{(b)}, W_i^{(b)})$ and $(V_i^{(t)}, W_i^{(t)})$ are the bottom and top $m \times m$ blocks of (V_i, W_i) , respectively. Similarly, $g_i^{(b)}$ and $g_i^{(t)}$ are the bottom and top m elements of each g_i . Once the much smaller reduced system is solved, via a dense system solver, the in this example three partitions $x_i, i = 1, 2, 3$, of the solution x are obtained as follows:

- (i) $x_1 = g_1 - V_1 z_2$
- (ii) $x_2 = g_2 - W_2 z_1 - V_2 z_4$
- (iii) $x_3 = g_3 - W_3 z_3$

Clearly, the above basic scheme may be made more efficient if in stage 2 one can generate only the bottom and top $m \times m$ tips of the spikes V_i and W_i , as well as the corresponding bottom and top tips of g_i . In this case, once the reduced system is solved, solving the system $Ax = f$ is reduced to solving the independent systems:

- (i) $L_1 U_1 x_1 = f_1 - B_1 z_2$
- (ii) $L_2 U_2 x_2 = f_2 - C_2 z_1 - B_2 z_4$
- (iii) $L_3 U_3 x_3 = f_3 - C_3 z_3$

If the matrix A is not diagonally dominant, we cannot guarantee that the diagonal blocks A_i are nonsingular. However, if we obtain the LU-factorization, without pivoting, of each A_i using diagonal boosting (perturbation), then

$$L_i U_i = (A_i + \delta A_i) \quad (4)$$

in which $\|\delta A_i\| = \mathcal{O}(\epsilon \|A_i\|)$ where ϵ is the unit roundoff. In this case, we will need to solve $Ax = f$ using an outer iterative scheme with the preconditioner being the matrix M which is identical to A except that each diagonal block A_i is replaced by $L_i U_i$ in Equation 4. Solving systems of the form $My = r$ is accomplished via the Spike scheme outlined above.

3.3 The Parallel PSPIKE Algorithm for Solving Linear Systems Arising in PDE-Constrained Optimization Problem

The linear systems that are extracted from the nonlinear solver has the following block structure

$$\begin{bmatrix} D & B^T \\ H & C^T \\ B & C & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \quad (5)$$

where $D \in \mathbb{R}^{n \times n}$ is diagonal and $D_{ii} > 0$ for $i = 1, 2, \dots, n$. Furthermore $H \in \mathbb{R}^{k \times k}$ is symmetric positive definite and $C \in \mathbb{R}^{k \times n}$ is dense with $k \ll m$. $B \in \mathbb{R}^{n \times n}$ is nonsymmetric banded and sparse within the band.

Premultiplying the above equation by $\begin{bmatrix} D^{-1} & & \\ & H^{-1} & \\ & & I \end{bmatrix}$ and rearranging the rows and columns the system can be reduced to

$$(H + J^T D J)x_2 = (f_2 - J^T f_1 + J^T D b) \quad (6)$$

where J and b are obtained by solving $BJ = C$ and $Bb = f_3$, respectively. Consequently, x_1 and x_2 can be computed via $x_1 = b - Jx_2$ and $x_3 = B^{-T}(f_1 - Dx_1)$. The solution process requires solving linear systems with the coefficient matrices B^T and B . We use the Krylov subspace method BiCGStab [11] with a banded preconditioner M to solve the systems $Ax = f$, in which the systems involving the preconditioner are solved via the PSPIKE scheme as described above. The LU-factorization of each diagonal block partition M_i (banded and sparse within the band) is obtained using PARDISO [10] without pivoting but utilizing diagonal boosting. In the next section we will illustrate the scalability of the above method for solving those systems that arise in biomedical PDE-constrained optimization problems.

4 Results

The distributed-memory test platform is a cluster with Infiniband interconnection. Each node has two Intel Harpertown processors where each processor contains four cores that run at 2.8GHz. The BiCGStab iterations for solving systems involving B^T and B are terminated when $\|\hat{r}_k\|_\infty / \|\hat{r}_0\|_\infty < \epsilon_{in}$ where $\hat{r}_0$ and $\hat{r}_k$ are the initial residual and the residual at k^{th} iteration, respectively, and the systems involving the preconditioners are solved via the PSPIKE scheme described above.

In this paper, we will describe a 3D hyperthermia model with over 1.8 million temperature variables in the NLP problem in (1). We consider linear systems extracted from the first, tenth (middle), and twenty first (last) iterations. However, since the results are uniform across these three systems, Figure 3 presents the results for the linear system of the tenth

Newton iteration.¹ Figure 3 depicts the speed improvement for an MPI/OpenMP hybrid implementation (8 cores(threads) per MPI processes) of the PSPIKE scheme. We demonstrate the speed improvement of the PSPIKE scheme for two stopping criteria, 10^{-5} and 10^{-7} . The corresponding final relative residuals are $\mathcal{O}(10^{-5})$ and $\mathcal{O}(10^{-7})$, respectively, which is sufficient to ensure the convergence of the inexact interior-point optimization process. For PARDISO and MUMPS the final relative residuals are $\mathcal{O}(10^{-12})$. Our new solver PSPIKE shows excellent scalability compared to these two state-of-the-art parallel solvers.

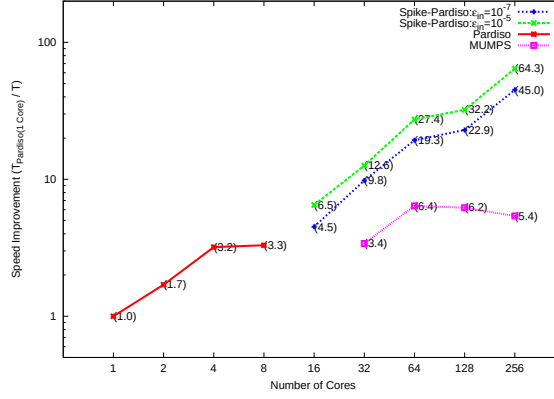


Fig. 3. The speed improvement of PSPIKE compared to PARDISO and MUMPS up to 256 cores. The base-line is the performance of PARDISO using one core (1'338 seconds).

5 Parallel Weighted Matching Algorithms

The most time consuming task solving large-scale PDE-constrained optimization problems is to solve the linear system $Ax = b$ by linear solvers (PSPIKE, MUMPS, PARDISO). When using pivoting in the direct solution of the solvers, as soon as a pivot element a_{ij} has been determined, no other element in row i or column j of A can be used as a pivot again. A greedy algorithm to choose the pivots can lead to sub-optimal results, may consume more storage and can dramatically increase the time for the solution [13]. Thus, of fundamental importance for the time behavior of the solver is to permute large entries to the diagonal of a sparse matrix. To do this, the pivoting is modelled as a weighted matching problem in a bipartite graph $G = (U, V, E)$.

Each row of A is represented by a vertex in U and each column by a vertex in V . A non-zero entry a_{ij} of A forms an undirected weighted edge between (i, j) . A subset $M \subseteq E$ in G is called a matching if no two edges of M are incident to the same vertex. In a maximum (cardinality) matching, no edge can be added to M without violating the matching property. An important concept in finding a maximum matching in bipartite graphs is that of an augmenting path. Let M be a matching in G . A path P in G is called M -augmenting if P has odd length, its ends are not covered by M , and its edges are alternatively out of and in M . Then, the symmetric difference $M \Delta P = (M \cup P) \setminus (M \cap P)$ is a matching and $|M \Delta P| = |M| + 1$.

A matching is called perfect if it is a maximum matching and every vertex of M is incident to a matching edge. It is not always possible to find a perfect matching. However, in our case the matrix is non-singular, thus there exists a perfect matching of G . The perfect matching M has cardinality n according to the $n \times n$ matrix A and defines an $n \times n$ permutation

¹ The solution of the linear system within the interior-point optimization process consumes more than $> 99\%$ of the overall time.

matrix $P = (p_{ij})$ with $p_{ji} = 1$ if $(i, j) \in M$ and a zero entry otherwise. The result of the perfect matching can be obtained by the matrices PA and AP with the matching entries on the zero-free diagonal. In a perfect weight matching, possible objectives are to maximize the sum or the product of the weights of the matched edges. The latter can also be transformed into a minimum-weight matching problem via logarithm scaling of the weights of the edges [14]. Then, the problem can be interpreted as the linear sum assignment problem (LSAP) of combinatorial optimization and thus, solution techniques of the LSAP can be applied to the matching problem as well [17].

Basically, linear solvers use two exact sequential algorithms. In order to find a maximum matching the algorithm of Hopcroft and Karp [16] with a worst case complexity of $\mathcal{O}(\sqrt{n}|E|)$ is implemented or in case of perfect weight matching the Hungarian method [15] with $\mathcal{O}(n(|E| + n \log n))$. However, the time complexity is still too high for large problems. Thus, the goal is to solve the weighted matching problem with a similar complexity in parallel in order to strengthen the parallel scalability of existing optimization methods. The goal is to obtain a perfect matching with maximum/minimum weight. Till today there are mainly three general concepts to solve the matching problem: approximation algorithms, primal-dual methods, and simplex-based algorithms. We will roughly outline the first and second method while focusing on the adaptation of the concepts to existing parallel algorithms.

5.1 Parallel Approximation Algorithms

While exact algorithms for weighted matching problems are pretty expensive to compute and hard to parallelize, approximation algorithms can be an efficient method to deal with large matching problems. In general, approximation algorithms are very fast, and nevertheless produce very good results even if these results are not optimal.

In this context the class NC is of fundamental importance. The class NC includes all problems which are computable in polylogarithmic time with a polynomially number of processors. The quality of an approximation algorithm for the weighted matching problem is measured by its approximation ratio. An approximation algorithm has an approximation ratio of ϕ , if it finds a matching with a weight of at least ϕ times the weight of an optimal solution for all graphs.

A $\frac{1}{2}$ -approximation algorithm can be obtained by the following greedy strategy. First, sort the weights of the edges in a decreasing order. Then, pick up the heaviest edge e , delete e and the incident edges, and repeat this process until the graph is empty. The sorting procedure results in a worst case complexity of $\mathcal{O}(|E| \log n)$. A linear-time implementation of this algorithm with a ratio of $\frac{2}{3} - \epsilon$ is provided by Preis [18]. Currently, the best known linear time approximation algorithm achieves an approximation ratio of $1 - \epsilon$ [19]. The main idea of the algorithm is to start with a matching and to allow each processor to make some local changes of this matching to improve its weight. To ensure that the local changes are independent of the changes of other processors the original graph is transformed into a new graph where an independent set of vertices corresponds to a set of pairwise independent local changes.

From the parallelization point of view the algorithm is inherently sequential. Thus, new parallel algorithms are emerging which are thereby trying to improve the time complexity and quality of the algorithm.

Manne et al. [20] present a parallel algorithm for distributed architectures which is very similar to the parallel calculation of the independent set [21]. The main idea is the use of dominating edges – edges that are heavier than their incident edges. In the algorithm, vertices are equally distributed across the processors and each processor will compute dominating edges. If the corresponding vertex of the edge does not reside on the same processor, communication is required to discover if the two vertices are allowed to be matched.

The algorithm has been applied to a problem with 5 million edges and 50000 vertices where it scales well with up to 32 processors. However, there are counter examples where the algorithm works sequentially. The worst case complexity of the algorithm is $\mathcal{O}(|E| \log |E|)$.

5.2 Parallel Primal-Dual Methods

Primal-dual algorithms operate with a pair of an unfeasible primal solution $x_{ij} \in \{0, 1\}$, $1 \leq i, j \leq n$, and a feasible dual solution u_i, v_j , $1 \leq i, j \leq n$, which fulfill the complementary slackness conditions:

$$x_{ij}(w_{ij} - u_i - v_j) = 0, \quad 1 \leq i, j \leq n$$

We denote $\bar{w}_{ij} := w_{ij} - u_i - v_j$ and call $\bar{w}_{ij}$ reduced costs with respect to the dual solution u_i, v_j . In general, these algorithms are focused on how to obtain a primal and dual solution which fulfills the complementary slackness conditions and how to implement the update of dual variables.

In the remainder of the paper, we will present parallel approaches based on the idea of augmenting path and auction algorithms.

Shortest Augmenting Path Finding the shortest augmenting path of a vertex requires a transformation of the original undirected weighted bipartite graph G in a directed weighted bipartite graph $\tilde{G} = (U, V, \tilde{E})$ where $\tilde{E}$ is filled with following directed edges. If $(i, j) \in M$, the corresponding edge is directed from i to j with weight 0 otherwise the edge is directed from j to i with weight of the corresponding reduced costs. Then, a free (not matched) vertex $s \in U$ is selected and the shortest path from s to all vertices of $\tilde{G}$ is computed. This problem is known as the single-source shortest path problem. The shortest among all paths from s to some free vertex in W is used to augment the current primal solution by swapping the free and the matched edges. The primal and dual solutions as well as the reduced costs are updated via $\delta = \min\{\bar{w}_{ij} : (i, j) \in E\}$ – the currently minimum uncovered reduced costs. After n augmentations an optimal primal solution is created. The overall complexity of the augmenting path algorithms amounts to $\mathcal{O}(n^3)$. Primal-dual parallel algorithms perform shortest path computation either starting from a single vertex and doing the shortest path computation from the single free vertex in parallel, or starting from multiple free vertices in order to execute multiple augmentation and variable updates in parallel. The former strategy will be outperformed by the latter one in the case of sparse problems because of the increase of the idle time of processors especially at the end of the algorithm. Recently, Madduri et al. [22] presented a very fast shortest path computation on a special shared memory architecture for certain graph families and problem sizes.

Auction Algorithms Auction Algorithms [23] might be suited for implementation on parallel machines and therefore, interesting for large-scale weighted matching problems. These algorithms simulate the auction process with bids, bidders, objects, and benefits of an object. U is interpreted as a set of persons i and V as a set of objects j . The weighted edge a_{ij} corresponds to the benefit which a person i obtains by buying the object j for free. Additionally, there is a price p_j , the dual variable, for each object j , and at each iteration unassigned persons bid simultaneously for objects with most benefit ($\max_{j \in V} \{a_{ij} - p_j\}$). Each person i increments the price p_{j_i} by a bidding increment $\gamma_i > 0$. Objects are awarded to the highest bidder. If an object remains without any bids during an iteration, its price and assignment status are left unchanged. This process terminates to a perfect matching in a finite number of iterations. If the perfect matching is a minimum weight matching, it highly depends on the bidding increment γ_i . If the increment γ_i is small enough to ensure the bid will be accepted, then the bid will be almost optimal for the bidder. Consequently,

the final matching will be almost optimal, too [23].

In particular, if the ϵ -complementary slackness condition (ϵ -CS)

$$\max_j \{a_{ij} - p_j\} - \epsilon \leq a_{ij_i} - p_{j_i} \quad \text{for all matched pairs } (i, j_i)$$

holds upon termination, the total benefit of the final matching is within $n\epsilon$ of being optimal. The standard method for choosing the bidding increment γ_i to fulfill the ϵ -CS condition is with

$$\gamma_i = \epsilon + d_i - e_i$$

where $d_i = \max_{j \in V} \{a_{ij} - p_j\}$ is the best objective value while e_i is the second best objective value. At the beginning of the algorithm ϵ is initialized with a large value and successively reduced up to a value which is less than the critical value $1/n$. This value is required to ensure $n\epsilon < 1$ and finally, to obtain a perfect matching.

Shared memory implementations of the auction algorithm can be divided into three classes: Jacobi parallelization, Gauss-Seidel parallelization, and a combination of the two mentioned before, called block Gauss-Seidel parallelization. We define a search task as the calculation of the best and second objective value, indicate s as the number of searcher processors used per person, and define p by the maximum available number of processors.

In the Jacobi parallelization the calculation for the bids per person i is assigned to a single processor ($s = 1$). The search tasks are executed in parallel and followed by merging the results of the search tasks of the other processors. As in our case, if the number of persons exceeds p then some processors are overloaded with multiple work. However, in the late stages of the method, processors are idle during the bidding phase and thereby reducing efficiency.

In the Gauss-Seidel parallelization the search task of an unassigned person is performed by p processors. Thus, the set of objects is divided into p pieces and all processors are assigned to the pieces doing the calculation for one person in parallel ($s = p$). After the search task is completed the results are merged together to figure out the best and second best objective value. The main drawback of this method is the large number of iterations arising in a huge number of persons such as in our case.

To overcome the drawbacks of the two approaches a hybrid method has evolved. Again, the bid calculation is parallelized as in the original Gauss-Seidel but the number of searcher processors used per bid is $1 < s < p$. With a proper choice of s , this method combines the best features of the other methods and alleviates their drawbacks. Most promising in the case of sparse matching problems are the Jacobi and hybrid methods. It is an open question if it is possible to transfer auction algorithms to message passing architectures in order to obtain a parallel scalability of the matching process.

6 Conclusion

We have presented a PDE-constrained interior-point algorithm for large-scale hyperthermia cancer treatment planning. The novel aspects of the approach is that we are using a globally convergent optimization method [5] and also a new scalable linear solver **PSPIKE**, which is designed to scale-up to thousands of cores. We have demonstrated that **PSPIKE** is an effective extension of **PARDISO** regarding parallel scalability on distributed-memory architectures. The experiments illustrate the computational advantages of our algorithm, and it is shown that the PDE-constrained optimization algorithm and implementation outperforms conventional approaches in terms of storage and CPU time for examples arising in an interesting emerging biomedical application. A major and challenging task is to parallelize the sequential matching strategy used in linear solvers in order to improve their parallel

scalability. Therefore, we have discussed parallelization approaches for weighted matching algorithms in bipartite graphs. In the future we would like to design parallel weighted matching algorithms for distributed architectures which enhance the parallel scalability of current nonlinear optimization solvers.

Acknowledgments. We thank Andreas Wächter (IBM Research), Esra Neufeld (IT'IS, ETH Zurich), Murat Manguoglu (University of Purdue), and Ahmed Sameh (University of Purdue) for useful comments and support.

References

1. Michael W. Berry and Ahmed Sameh. Multiprocessor schemes for solving block tridiagonal linear systems. *The International Journal of Supercomputer Applications*, 1(3):37–57, 1988.
2. G. Biros and O. Ghattas. Parallel Lagrange-Newton-Krylov-Schur methods for PDE-constrained optimization. Part I: The Krylov-Schur solver. *SIAM Journal on Scientific Computing*, 27(2):687–713, 2005.
3. G. Biros and O. Ghattas. Parallel Lagrange-Newton-Krylov-Schur methods for PDE-constrained optimization. Part II: The Lagrange-Newton solver and its application to optimal control of steady viscous flows. *SIAM Journal on Scientific Computing*, 27(2):714–739, 2005.
4. R. H. Byrd, F. E. Curtis, and J. Nocedal. An inexact Newton method for nonconvex equality constrained optimization. *Mathematical Programming, Series A*, 2009. (To appear).
5. F. E. Curtis, O. Schenk, and A. Wächter. An Interior-Point Algorithm for Large-Scale Nonlinear Optimization with Inexact Step Computations. *SIAM Journal on Scientific Computing*, 2009. (Submitted).
6. E. Haber and U. M. Ascher. Preconditioned all-at-once methods for large, sparse parameter estimation problems. *Inverse Problems*, 17(6):1847–1864, 2001.
7. Esra Neufeld. *High Resolution Hyperthermia Treatment Planning*. PhD thesis, ETH Zurich, August 2008.
8. Harry H. Pennes. Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm. *J Appl Physiol*, 1(2):93–122, 1948.
9. Eric Polizzi and Ahmed H. Sameh. Spike: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113–120, 2007.
10. Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475 – 487, 2004. Selected numerical algorithms.
11. H. A. van der Vorst. Bi-cgstab: a fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, 1992.
12. A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.
13. Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22, 973–996, 1999.
14. Markus Olschowa and Arnold Neumaier. A New Pivoting Strategy For Gaussian Elimination. *Linear Algebra and Its Application*, 240, 131–151, 1996.
15. H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2, 83–97, 1955.
16. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Computing*, 2, 225 – 231, 1973.
17. R. E. Burkard and E. Cela. Linear assignment problems and extensions. In P. M. Pardalos and D. Z. Du, editors, *Handbook of Combinatorial Optimization - Supplement Volume A*, pages 75–149. Kluwer, 1999.
18. R. Preis. Linear time $1/2$ -approximation algorithm for maximum weighted matching in general graphs. In *Proceedings of STACS 1999*, C. Meinel and S. Tison, Eds. Lecture Notes in Computer Science, vol. 1563. Springer-Verlag, New York, 259–269, 1999.
19. Stefan Hougardy and Doratha E. Vinkemeier. Approximating weighted matchings in parallel. *Information Processing Letters*, 99(3): 119–123, 2006.
20. F. Manne and R. H. Bisseling. A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem. In *Proc. Seventh Int. Conf. on Parallel Processing and Applied Mathematics (PPAM 2007)*, Lecture Notes in Computer Science, Vol. 4967, Springer-Verlag, Berlin, pp. 708–717, 2007.
21. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comp.* 15, 1036–1053, 1986.
22. Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joe R. Crobak. Parallel Shortest Path Algorithms for Solving Large-Scale Instances. In C. Demetrescu, A.V. Goldberg, and D. Johnson, editors, *Shortest Path Computations: Ninth DIMACS Challenge*, American Mathematical Society, 2008, to appear.
23. Dimitri P. Bertsekas and David A. Castañón. Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Computing* 17, 707–732, North Holland, 1991.

Mesosopic Simulations of Hydrodynamic Interactions Using a Particle-Based Model on Cell Broadband Engine

Annika Schiller and Godehard Sutmann

Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre, Research Centre Jülich, Germany
{a.schiller,g.sutmann}@fz-juelich.de

Abstract. In hydrodynamics, mesoscopic simulations are important for the understanding of the interplay between macroscopic flow and microstructures. One mesoscopic simulation technique is Multiparticle Collision Dynamics (MPC), a particle-based approach which bases on a free streaming of the particles and a stochastic rotation of the particle velocities. To accelerate the MPC algorithm, the Cell Broadband Engine (Cell/BE) is considered here as target platform for a parallel implementation. The Cell/BE is a heterogeneous multicore processor consisting of multiple heterogeneous execution units, SIMD processing engines, fast local storages and a software-managed memory-hierarchy.

An implementation of the MPC algorithm for Cell/BE is developed on an application-oriented level by using the high-level programming model CellSs (Cell Superscalar Framework). In this work, the MPC algorithm is tested for its ability to exploit the Cell/BE specific features by porting the two basic steps which are embarrassingly parallel. The steps of porting the MPC algorithm via CellSs are pointed out and the performance results obtained by the calculation on the Cell/BE are discussed.

Key words: particle simulations; multi-core architectures; Cell Broadband Engine; hydrodynamics; parallel computing

1 Introduction

The simulation of hydrodynamic properties of fluid and flow phenomena on the mesoscale is a challenge for conventional techniques, e. g. Molecular Dynamics or Monte Carlo. In many cases, the mesoscopic behavior of fluids or flows is important to understand as it is key to the interplay between macroscopic flow and microstructures. To overcome the wide range of length and time scales, several mesoscale simulation techniques were developed. Their principle is to coarse-grain out irrelevant atomistic details while correctly incorporating the essential physics and conservation laws. One of these mesoscale simulation techniques is the so called Multiparticle Collision Dynamics (MPC), which solves the linearized Navier-Stokes equations by using a particle-based approach.

Since the requirement of high performance computing and memory for computer simulations is growing, hardware architecture determines the efficiency of a simulation in terms of execution time. High performance computing (HPC) clusters have been commonly used to study scientific computing applications. They provide high performance by splitting the computational tasks among the nodes in the cluster. Due to hardware constraints like power consumption, heat dissipation and other physical limitations the development trend more and more tends to multicore design patterns. The Cell Broadband Engine (Cell/BE) is a heterogeneous multicore processor and has been designed to support a broad range of applications. It combines considerable floating-point resources required for demanding numerical algorithms with a power-efficient software-controlled memory hierarchy [6]. The Roadrunner at the Los Alamos National Laboratory is a hybrid of General Purpose CISC Opteron as well as Cell/BE processors. The fact that Roadrunner is the first supercomputer to run at PetaFlops speeds, indicates that the Cell/BE is a promising architecture [10].

To simplify the enabling of applications on the Cell/BE, high-level programming models were developed. One example is the Cell Superscalar Framework (CellSs) [11] which offers a portable programming model to port, parallelize and tune applications on Cell/BE. CellSs

allows the programmer to write sequential code based on which CellSs is able to exploit the existing concurrency and to use the different components of the Cell/BE (PPE and SPEs) by means of an automatic parallelization at execution time.

In this work, the two basic steps of MPC are ported to Cell/BE to examine the potential and capabilities of the Cell/BE particularly with regard to MPC. The main purpose is to develop a Cell/BE implementation on an application-oriented level. That is not to produce highly optimized code using the native programming API, but furthermore to invest the amount of time that scientists developing applications are usually willing to spend and to achieve acceptable performance. Hence, ease of programming and resulting performance of CellSs shall be assessed, to find out whether it is a useful alternative to native programming.

2 Multiparticle Collision Dynamics

Multiparticle Collision Dynamics (MPC) is a particle based method which captures hydrodynamic effects on a mesoscopic scale and generalizes the concept of Direct Simulation Monte Carlo, which takes into account binary collisions between particles [3]. As a characteristic, it conserves mass, momentum and energy and captures both full hydrodynamic interactions and thermal fluctuations. Various methods of MPC exist, which differ in the way, how the collision step is performed [4]. Extensions of MPC may also conserve angular momentum, but in the present work, the original version, called Stochastic Rotation Dynamics (SRD), is implemented.

MPC, was first introduced by Malevanets and Kapral in 1999 [1]. The fluid is modeled by particles whose positions and velocities are continuous variables but the system evolves in discrete time steps. It is important to mention that the particles in this method do not represent the particles of the fluid, but can rather be interpreted as momentum and energy carriers which represent the system's properties.

The evolution of the system consists of two basic steps: a streaming step and a collision step. In the streaming step, the particles perform a uniform, linear motion without interacting with each other. Hence, the particle positions are updated according to

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t \quad (1)$$

where Δt is the length of one time step, $\mathbf{r}_i(t)$ is the position of particle i at time t and $\mathbf{v}_i(t)$ the velocity of particle i at time t .

The interactions among the particles are considered in the collision step. To perform a collision between multiple particles, a regular grid is superimposed over the geometrical domain of the simulation box. The grid spacing a is assumed to be unit and constant during time. Each particle is assigned to one grid cell and collisions are then modeled by a simultaneous stochastic rotation of the relative velocities of the particles in each cell. For that purpose, the centre of mass velocity $\mathbf{v}_{cm,c}$ of each cell is calculated via $\mathbf{v}_{cm,c} = (1/N_c) \sum_{i=1}^{N_c} \mathbf{v}_{i,c}$, where N_c is the number of particles in cell c and $\mathbf{v}_{i,c}$ are the particle velocities in the cell. The particle velocities are then transformed in each cell as follows

$$\mathbf{v}_{i,c}(t + \Delta t) = \mathbf{v}_{cm,c}(t) + \mathbf{R}_c\{\alpha; \hat{n}_\xi\} [\mathbf{v}_{i,c}(t) - \mathbf{v}_{cm,c}(t)] \quad (2)$$

In this equation, $\mathbf{R}_c$ is a stochastic rotation matrix defined for cell c , which performs a rotation of a given angle α around a randomly chosen rotation axis $\hat{n}_\xi$, which is independently chosen in each collision cell. The rotation angle α is defined at the beginning of the simulation. It is the rotation angle α , the particle density ρ and the time step Δt which determine the fluid properties, e.g. viscosity or diffusivity [4].

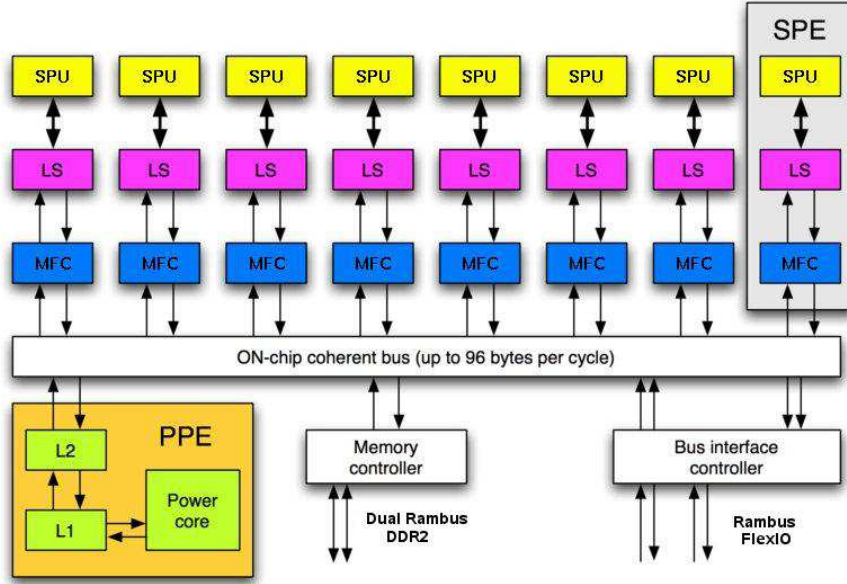


Figure 1: Schematic of the architecture of the Cell/BE

In order to conserve Galilean invariance, the grid is randomly displaced by a shift-vector $\xi \in [-\frac{a}{2}, \frac{a}{2}]$ in each time step [5]. The grid shift ensures the principle of “molecular chaos” on a length scale a . Shifting the grid at each time step avoids that the particles have memory of previous collisions, which contradicts the fact that collisions are uncorrelated in space and time. Such memory effects can occur, when particles remain in the same cell from one time step to the next and therefore collide with the same particles for a large number of time steps.

3 The Cell Broadband Engine

The Cell/BE was developed by STI, a consortium formed by Sony, Toshiba and IBM in 2001. It is a single-chip multi-processor with nine heterogeneous processor cores operating on a shared, coherent memory. The principle of this architecture is to combine a general-purpose PowerPC architecture core of modest performance with streamlined co-processing elements, which greatly accelerate multimedia and vector processing applications, as well as many other forms of dedicated computation. The Cell/BE has a theoretical peak performance of 204.6 GFLOPS [7].

3.1 Cell/BE Architecture

The processors are grouped into two types: one PowerPC element (PPE) and eight so-called synergistic processing elements (SPEs). The PPE is a general purpose, dual-threaded, 64-bit RISC processor and complies with the 64-bit PowerPC architecture. It is optimized for control-intensive tasks. Therefore, the PPE is responsible for the overall control of the system. It runs the operating system for all applications running on the Cell/BE and manages system resources. The PPE is also responsible for the allocation and management of SPE threads [7].

The SPEs are optimized for running compute-intensive applications, primarily accelerating media and streaming workloads. They are independent vector processors operating on 128 registers, each 128 bit wide. The SPEs are designed to be programmed in high-level languages and support a rich instruction set that includes SIMD extensions. Each SPE contains

a 256 kByte embedded SRAM for instruction and data, which is called Local Store (LS). The LS is visible to the PPE and can be addressed by software directly. The SPEs cannot access main memory directly but only operate on their own LS, i.e. they fetch instructions from their LS, and loads and stores data from and to their LS. One SPE has a theoretical peak performance of 25.6 GFLOPS at 3.2 GHz [7].

The Element Interconnect Bus (EIB) realizes a high-bandwidth, memory-coherent connection between all components of the Cell processor including the PPE, the SPEs, the main memory and I/O. The EIB is implemented as a circular ring comprised of four 16-byte-wide uni-directional channels. Its theoretical peak bandwidth is 204.8 GByte/s [9].

3.2 Cell/BE Programming

Before a programmer ports an application to the Cell/BE, he should carefully reflect if the application is well-suited to run on Cell/BE. On any processor, the best performance optimizations are achieved by exploiting the specific features of the hardware. The Cell/BE specific features are the large number of SPEs, their large register file, and their ability to hide main-storage latency by concurrent computation and DMA transfers. In general, one can say that an application achieves good performance on Cell/BE if it follows the single-instruction, multiple data (SIMD) strategy and does not make heavy use of branches and scalar operations [6].

Basically, there are two main ways to program applications for the Cell/BE: (1) the native programming approach and (2) the high-level programming model approach. The native programming bases on the Cell Software Development Kit (Cell SDK) [8]. The Cell SDK is a full package of tools for the creation of applications for the Cell/BE. It is composed of runtime tools such as the Linux kernel, development tools, software libraries and frameworks, performance tools, a full system simulator, and example source files. With the Cell SDK, separate source codes have to be developed for PPE and SPEs, which implies that the code can only run on Cell/BE. The syntax of the SDK bases on C/C++ language extensions to generate threads, to initiate DMA transfers et cetera [8]. With the Cell SDK, all actions like DMA transfer, double-buffering and the distribution of work among the SPEs have to be programmed explicitly. Consequently, the responsibility completely lies at the programmer providing high flexibility and great challenges at the same time.

To simplify the enabling of applications on the Cell/BE, several high-level programming models were developed. One example is the Cell Superscalar Framework (CellSs) [11] which offers a portable programming model to port, parallelize and tune applications on Cell/BE. CellSs is developed and still enhanced by Barcelona Supercomputing Centre (BSC). With CellSs, only one source file has to be written, which contains both the code for the PPE and the SPEs. The CellSs syntax bases on simple annotations (pragmas) that are inserted into the application code to specify the parts of the program that are intended to run on the SPEs [12]. The advantage is, that the CellSs code can also be run on other architectures, because the pragmas are ignored by other compilers. From the pragma equipped source code CellSs generates code for PPE and SPEs and many process stages like the scheduling of independent tasks to different SPEs, the required DMA transfers as well as double-buffering are handled automatically by the CellSs compiler so they do not have to be programmed explicitly [12]. This greatly facilitates the programming of applications for Cell/BE. However, with this ease not only the responsibility but also the flexibility is taken from the programmer, which can have negative effects on the performance.

4 MPC on Cell/BE

To accelerate the MPC algorithm by using the Cell/BE, its architectural requirements including multiple heterogeneous execution units, SIMD processing engines, fast local stores and a software managed cache, need to be considered. In general, the MPC algorithm should be an ideal candidate to be ported to Cell/BE. The algorithm is simple and the SIMD engines of the Cell/BE can be exploited as the same instructions have to be applied to a large amount of data. The MPC algorithm also provides a high level of parallelism as the operations applied to the particles of one grid cell are completely independent from the operations applied to particles of another grid cell.

To prove the theory that MPC is well suited to be ported to Cell/BE, the free streaming step and the multiparticle collision step were chosen to be calculated on the SPEs in parallel in a first step. If a good performance can be achieved with these two steps on Cell/BE, i. e. if the resulting speedup is close to linear scaling, it is worth the effort to also parallelize the cell filling step. This step is more complicated to parallelize as there are many dependencies among the operations. The free streaming step and the multiparticle collision step are embarrassing parallel because the operations are cellwise independent. However, the most challenging task when porting these steps to Cell/BE is to achieve data locality so that the information belonging to one grid cell are in consecutive memory. This is important, because the data has to be transferred via DMA into the local stores of the SPEs before they can start their calculation.

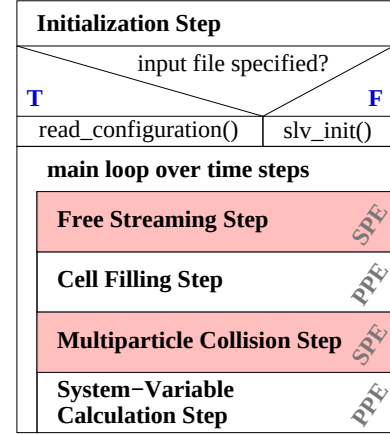


Figure 2: Program flow of the MPC algorithm

4.1 Steps of Porting

For porting the MPC algorithm to Cell/BE, the high-level programming model CellSs is used, as ease of programming is an important aspect when opening a new hardware architecture to a broad range of applications. Our purpose is to port MPC on an application-oriented level because scientists developing applications are usually no hardware specialists and are therefore not able or do not have the time to produce highly optimized CellSDK implementations.

Basically, the work of parallelizing an application involves the following tasks [6]:

- Distribution of work among processors
- Distribution of data if memory is distributed
- Synchronization of subtasks
- Transfer of data if memory is distributed

Distribution of work For the distribution of work among the used SPEs, the Function-Offload Model [6] was chosen. In this model, the SPEs are used as accelerators, so that time-consuming parts of the application are offloaded onto the SPEs. Furthermore, every SPE executes the same source code and the data is distributed among the SPEs. In the current implementation of the MPC algorithm, the free streaming step and the multiparticle collision step were determined to be the parts of the algorithm that are time-consuming and above all easy to parallelize.

Distribution of data As it was decided to use CellSs to port the MPC algorithm to Cell/BE, the distribution of data is done automatically by the CellSs runtime. The CellSs runtime library builds a data dependency graph based on which it decides which operations can be executed in parallel and how the work is distributed [12]. Hence, the programmer can influence the parallelism and thereby the efficiency by ensuring the data to be independent. In the current implementation, the operations and hence the data are completely independent, i.e. for the calculation of the new positions and velocities of particles in a grid cell, only the information of particles in this cell is used. There is no access to the information of particles in other cells.

Synchronization of subtasks The synchronization of subtasks is also done automatically by CellSs on the basis of the pragmas. The tasks are scheduled according to the data dependency graph. If the data of two tasks is independent, both tasks can be submitted to different SPEs and executed in parallel. The programmer only has to take care of waiting on data. That is of importance if a program part that runs on the PPE operates on data which is calculated in parallel by the SPEs previously. In this case the programmer has to set a barrier to ensure that the calculation on the SPEs has finished and the required data is available. In the current implementation, barriers are necessary after the execution of the free streaming step and the multiparticle collision step which are intended to run in parallel on the SPEs. These parallel program parts are followed by sequential program parts on the PPE (see Figure 2).

Transfer of data The transfer of data is probably the most difficult point of the current implementation. With CellSs, the data transfer is realized through the parameters of an annotated function that is intended to run on SPE. With the task specification pragma, the size and the direction of the data transfer is specified [12]. This requires that the information of particles in one grid cell lies in consecutive memory. In the serial implementation linked-cell lists are used to assign the particles to the grid cells. The advantage is, that linked-cell lists are very efficient in memory requirement and computation. They basically consist of two integer arrays pointing to each grid cell and each particle within the cell, so that the memory requirement grows linearly with the number of particles. The linked-cell lists can be set up very efficient because the particle information reside at the same place in memory, only the integer arrays are updated. The efficient update of the lists is an important aspect because they have to be updated every time step. However, the drawback of the linked-cell lists is that the information of particles in one grid cell is not in consecutive memory. Therefore, it is not possible to transfer the information of one grid cell to the SPEs in one block.

To achieve cellwise data locality, the data management was changed from linked-cell lists to a strategy where the particle information is directly stored cellwise. In this strategy, memory is allocated for each grid cell and if a particle changes the cell, the associated particle information needs to be restored. As the number of particles in one grid cell does not stay constant during time but fluctuates around the average number of particles in a cell, some redundant memory needs to be allocated for each grid cell. To fit into the local store of the SPEs which is limited to 256 kB, several neighboring grid cells are combined to one block of data which is then transferred to the SPEs at once. The drawback of this strategy is, that for each grid cell the whole allocated memory, which also includes the parts that are not filled with particle information, is transferred to the SPEs.

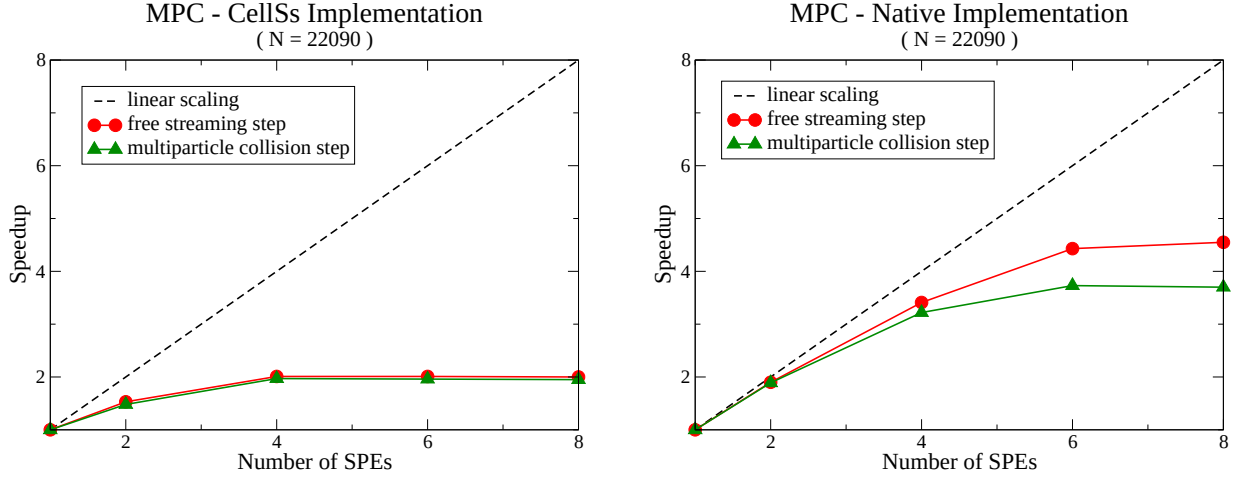


Figure 3: The figure shows the speedup of the free streaming step and the multiparticle collision step on the Cell/BE using CellSs (left) on the one hand and Cell SDK (right) on the other hand.

5 Performance Results

The left of Fig. 3 shows the first performance results for the free streaming step and the multiparticle collision step. It is visible, that the speedup levels off at 2 which is far off linear scaling. A comparable implementation using Cell SDK shows the same effect (see right of Fig. 3). Although the speedup levels off at around 4 in this case, it is nevertheless still no linear scaling. The left of Fig. 4 shows the execution time comparison between CellSs and Cell SDK implementation of the free streaming step and the cell filling step. For one SPE, the execution time of the CellSs and Cell SDK implementation is nearly the same for both steps, which verifies that both implementations are based on the same algorithm. With the use of more SPEs, the execution time of the Cell SDK implementation decreases more rapidly than the execution time of the CellSs implementation.

A look at the visualized traces of the CellSs implementation gives information about the reasons for the bad scaling. The upper picture of Fig. 5 shows the trace visualization of one iteration of the CellSs implementation using 4 SPEs. The first block of bars represent the execution time of the free streaming step on the SPEs and the second block shows the execution time of the multiparticle collision step. The black areas in between the bars indicate, that the SPE is idle during that time. The traces show that the speedup cannot be linear, because the SPEs are idle many times, i. e. the execution of the free streaming step and also the multiparticle collision step is not fully parallel. A comparison between the traces of different numbers of SPEs will prove that the idling times increase for a larger number of SPEs which stresses the fact that the speedup levels off at a certain value.

A visualization of the task stages in detail by zooming into the traces clarifies the reason for the bad loadbalancing. It can be seen, that the communication, i. e. the DMA transfer of data from main memory to the local store of the SPEs, strongly dominates over the computation. The granularity of the parallelized program parts is too small, i. e. the amount of operations needed for the free streaming step and the multiparticle collision step is too small compared to amount of DMA transfer. In this case, the communication cannot be overlapped completely by the computation and therefore limits the performance gain with an increasing number of SPEs. This effect has a stronger influence on the CellSs implementation because in contrast to the native implementation, the CellSs runtime produces some additional overhead for the scheduling of tasks. This overhead does not carry weight in implementations with large granularity, but can have strong effects on the performance in communication intensive applications with low granularity as can be seen in the current example.

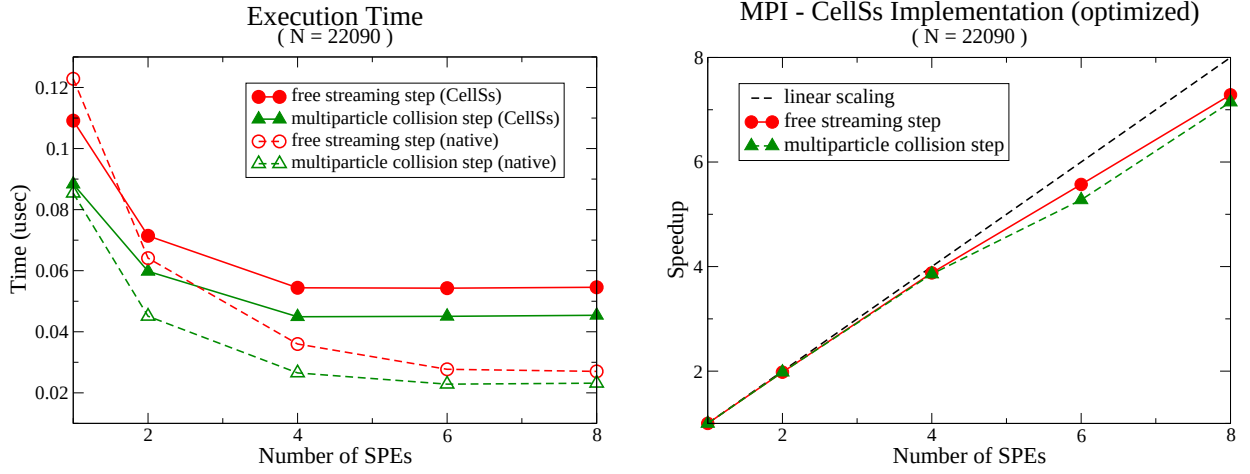


Figure 4: In the left, the execution time of the free streaming step and the multiparticle collision step obtained with CellSs is compared to the execution time achieved with the Cell SDK implementation. In the right, the speedup of the optimized CellSs implementation of the two basic MPC steps is shown.

5.1 Optimization of Data Transfer

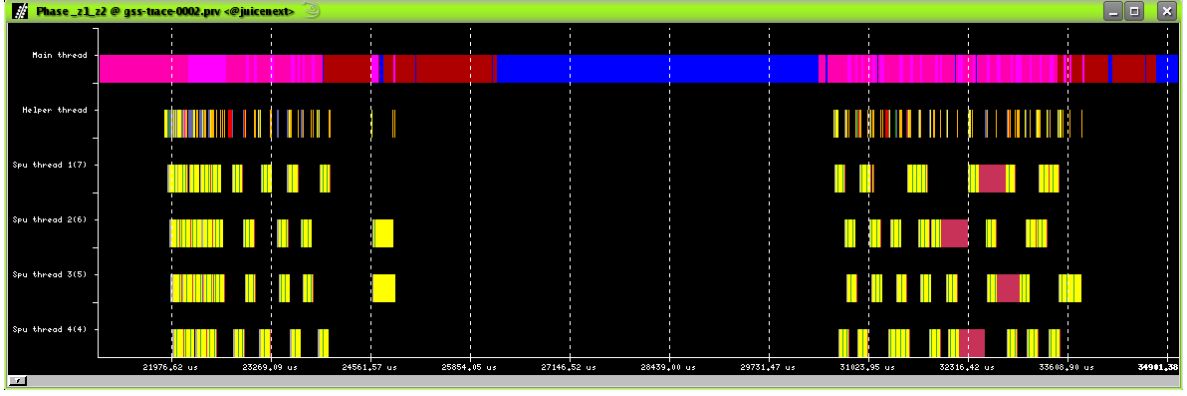
To improve the performance, the granularity of the parallelized program parts has to be increased or the amount of DMA transfer has to be reduced. An increase of the granularity can be achieved by parallelizing the cell filling step, so that the whole iteration runs in parallel on the SPEs. However, this is time consuming and therefore will only be realized if MPC turns out to be a suitable application for the Cell/BE. The reduction of the DMA transfer can be achieved easier, by minimizing the memory allocated for one grid cell.

The empty memory of each collision cell which is also transferred to the SPEs can be minimized by choosing a different strategy for exchanging the particles. In the first version the particle information is restored without using an additional buffer, so that the particle information is directly moved to the memory location of the new collision cell. If the memory of this cell is too small to hold additional information, the memory is increased. However, the memory is not decreased afterwards if particles leave this cell, because reallocation of memory is costly and should only be executed a few times. To minimize the empty memory locations, an additional buffer is allocated to hold the complete particle information. When the particles are assigned to the grid cells, their information is inserted into the second buffer instead of moving them to a different location in the same buffer. Hence, as the number of particles in a cell fluctuates around the average number of particles in a cell, the empty memory locations can be minimized.

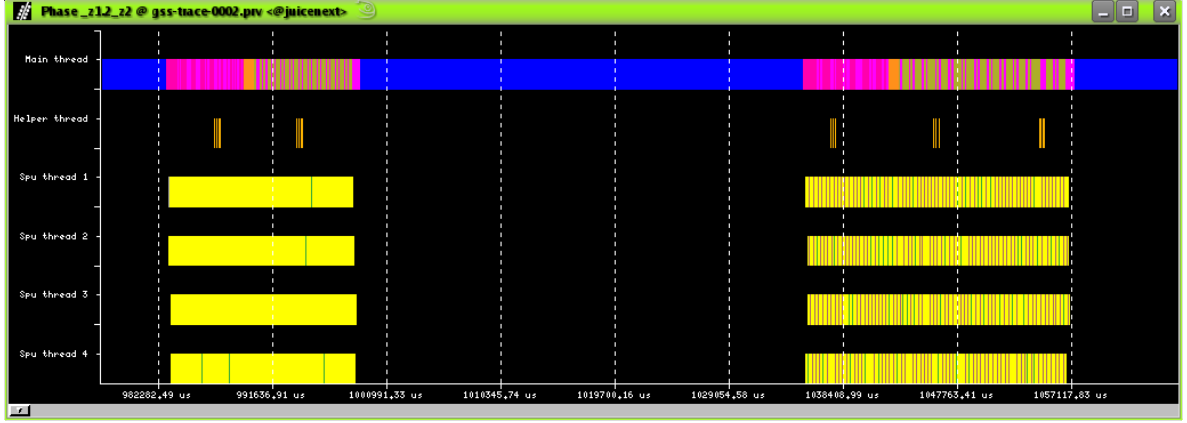
In the right of Fig. 4, the performance results for the optimized data transfer is displayed. It can be seen, that the speedup is now close to linear scaling for both steps. A look at the visualized tracefiles (Fig. 5 (b)) shows that the execution of the parallelized program parts on the SPEs is now fully parallel without idling times. Zooming into the traces to see the task stages will prove that the amount of data transfer was reduced so that the communication can be overlapped completely by the computation now. The granularity of the two parallelized program parts is still small, but with the parallelization of the cell filling step, the granularity will be increased which will also have positive effects on the performance.

6 Conclusions and Future Work

With the objective of porting the MPC algorithm to Cell/BE on an application-oriented level, the high-level programming model CellSs was used to develop a Cell/BE implemen-



(a) trace visualization of original CellSs implementation



(b) trace visualization of optimized CellSs implementation

Figure 5: The above figures show extracts of the tracefile visualization with Paraver [13] of program runs on Cell/BE using 4 SPEs. The topmost bar displays the execution time of the main thread, which corresponds to the PPE main thread. The second bar represents the PPE helper thread, that schedules tasks for the execution on the SPEs. Depending on the number of used SPEs, the lower bars depict the execution on the SPEs where one bar represents one SPE. All extracts show one iteration, where the first block of SPE threads represents the free streaming step and the second block the multiparticle collision step.

tation on the basis of MPC. To find out if MPC is a suitable application for the Cell/BE, the two basic steps of MPC, the free streaming step and the multiparticle collision step, are calculated in parallel on the SPEs in a first step. After an improvement of the DMA transfer, where the communication is completely overlapped by computation, a nearly linear scaling was achieved. This shows, that the Cell specific features can be exploited with the MPC algorithm so that it is reasonable to also port the cell filling step to the Cell/BE.

It has to be mentioned that the time period of code development for the Cell/BE was too long for an application-oriented level. Compared to the native programming API (Cell SDK), the Cell/BE implementation of application codes is strongly facilitated with the programming model CellSs. This is mainly because many process stages are invoked automatically by the CellSs runtime and consequently the development phase is expected to be significantly shorter. However, during the development phase several problems concerning the CellSs implementation on our Cell cluster occurred which could finally be bypassed but which are not yet fully understood. Additionally, debugging with CellSs turned out to be difficult as CellSs does not give insight into the internal processes. CellSs is nevertheless a promising programming model for the Cell/BE because it is easy to learn due to its similarity to OpenMP. Therefore, the cell filling step will also be ported via CellSs with the aim to significantly reduce the development phase.

For the parallelization of the cell filling step a domain decomposition strategy will be applied grouping neighboring collision cells into parts that fit into the local store of the SPE. In this case, a return to linked-cell lists will be reasonable having one linked-cell list for each SPE domain. With this strategy the granularity of the parallelized parts of the program will be increased because more algorithmical tasks are calculated in parallel with reduced intermediate data transfer. An important issue one have to deal with will be the exchange of data between the different domains because the particles move through the simulation system.

The results obtained in this work show once more that it is challenging to program applications for Cell/BE even when using high-level programming models that should simplify the porting. Nevertheless, to simplify the programming it is necessary to support the development of programming models like CellSs by using them and hence giving developers the possibility to enhance them with regard to user remarks.

Acknowledgments. We would like to thank BSC, namely Rosa Badia and Luis Martinell for their support and helpful discussions.

References

1. A. Malevanets and R. Kapral. Mesoscopic model for solvent dynamics. In *Journal of Chemical Physics*, volume 110, pages 8605–8613, May 1999.
2. I.O. Götz, H. Noguchi and G. Gompper. Relevance of angular momentum conservation in mesoscale hydrodynamics simulations. In *Phys. Rev. E*, volume 76, pages 046705, 2007.
3. G.A. Bird. *Molecular Gas Dynamics*. Clarendon, Oxford, 1976.
4. G. Gompper, T. Ihle, D.M. Kroll and R.G. Winkler. Multi-Particle Collision Dynamics: A Particle-Based Mesoscale Simulation Approach to the Hydrodynamics of Complex Fluids. In *Adv. Polym. Sci.*, volume 221, pages 1 – 87, 2009.
5. T. Ihle and D.M. Kroll. Stochastic rotation dynamics. A Galilean invariant mesoscopic model for fluid flow. In *Phys. Rev. E*, volume 63, pages 020201, 2001.
6. A. Arevalo, R. M. Matinata, M. Pandian, E. Pery, K. Ruby, F. Thomas and C. Almond. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. ISBN: 0738485942, Redbooks, 2008.
7. IBM Cell Broadband Engine: Programming Tutorial. Version 2.1, March 2007.
8. IBM Software Development Kit for Multicore Acceleration: Programmer's Guide. Version 3.1, November 2008.
9. M. Bolten, A. Dolfen, N. Eicker, I. Gutheil, W. Homberg, E. Koch, A. Schiller, G. Sutmann and L. Yang. JUICE – Jülich Initiative Cell Cluster. Internal report, FZJ-JSC-IB-2007-13, Jülich Supercomputing Centre, December 2007.
10. Ken Koch. Roadrunner Platform Overview. In *Roadrunner Technical Seminar Series*, Los Alamos National Laboratory, March 2008.
11. P. Bellens, J. M. Perez, R. M. Badia and J. Labarta. CellSs: a programming model for the cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.
12. Barcelona Supercomputing Centre Cell Superscalar (CellSs) User's Manual. Version 2.2, May 2009.
13. Paraver - Parallel Program Visualization and Analysis tool *Version 2.1*, November 2000.

HICFD – Hocheffiziente Implementierung von CFD-Codes für HPC-Many-Core-Architekturen

Thomas Alrutz⁴, Petra Aumann⁷, Achim Basermann³, Kim Feldhoff⁵, Thomas Gerhold¹, Jörg Hunger⁸, Jens Jägersküpper¹, Hans-Peter Kersken³, Olaf Knobloch⁷, Norbert Kroll¹, Olaf Krzikalla⁵, Edmund Kügeler², Ralph Müller-Pfefferkorn⁵, Mathias Puetz⁶, Andreas Schreiber³, Christian Simmendinger⁴, Christian Voigt², and Carsten Zscherp⁸

¹ Deutsches Zentrum für Luft- und Raumfahrt e.V., Institut für Aerodynamik und Strömungstechnik, C²A²S²E: Center of Computer Applications in Aerospace Science and Engineering, Bunsenstr. 10, D-37073 Göttingen, {thomas.gerhold,jens.jaegerskuepper,norbert.kroll}@dlr.de

² Deutsches Zentrum für Luft- und Raumfahrt e.V., Institut für Antriebstechnik, Linder Höhe, D-51147 Köln, {edmund.kuegeler,christian.voigt}@dlr.de

³ Deutsches Zentrum für Luft- und Raumfahrt e.V., Simulations- und Softwaretechnik, Linder Höhe, D-51147 Köln, {achim.basermann,hans-peter.kersken,andreas.schreiber}@dlr.de

⁴ T-Systems Solutions for Research GmbH, Bunsenstr. 10, D-37073 Göttingen, {thomas.alrutz,christian.simmendinger}@t-systems-sfr.com

⁵ Technische Universität Dresden, Zentrum für Informationsdienste und Hochleistungsrechnen, Zellescher Weg 12, D-01062 Dresden, {kim.feldhoff,olaf.krzikalla,ralph.mueller-pfefferkorn}@tu-dresden.de

⁶ IBM Deutschland GmbH, Hechtsheimer Str. 2, D-55131 Mainz, mathias.puetz@de.ibm.com

⁷ Airbus Deutschland GmbH, Aerodynamic Tools and Simulation, {petra.aumann,olaf.knobloch}@airbus.com

⁸ MTU Aero Engines GmbH, Dachauer Str. 665, D-80995 München, {joerg.hunger,carsten.zscherp}@mtu.de

Zusammenfassung Bei dem Forschungsprojekt HICFD handelt es sich um ein Verbundprojekt des vom Bundesministerium für Bildung und Forschung geförderten Programms „IKT 2020 – Forschung und Innovation“. Das Forschungsprojekt hat zum Ziel, neue Methoden und Werkzeuge zur Analyse und Optimierung des Leistungsvermögens strömungsmechanischer, paralleler Programme auf Hochleistungsrechnern mit Prozessoren mit einer Vielzahl von Kernen zu entwickeln und diese exemplarisch auf die strömungsmechanischen Codes TAU und TRACE des Projektpartners DLR anzuwenden. Die wesentlichen Ziele des Projektes werden in dieser Arbeit vorgestellt.

Das Leistungsvermögen strömungsmechanischer, paralleler Programme soll in diesem Projekt durch eine optimale Ausnutzung aller Parallelitätsebenen verbessert werden. Auf der obersten Ebene (MPI) soll eine intelligente Gitteraufteilung den Lastausgleich zwischen den MPI-Prozessen verbessern. Für blockstrukturierte Gitter soll hier ein Multi-Core-kompatibles Partitionierungswerkzeug entwickelt werden. Auf der Ebene der Multi-Core-Architektur sollen exemplarisch für die beiden Strömungslöser TAU und TRACE hochskalierende hybride OpenMP/MPI-Verfahren implementiert werden. Auf Prozessorkern-Ebene soll ein Präprozessor entwickelt werden, der die komfortable Nutzung paralleler SIMD-Einheiten („Single Instruction Multiple Data“) auch für komplexe Anwendungen ermöglicht. Um zu überprüfen, in welchem Umfang die Anwendung des Präprozessors das Leistungsvermögen paralleler Programme beeinflusst, soll die Leistungsanalyse-Software Vampir in Richtung SIMD erweitert werden.

Key words: Performance-Analyse, Vampir-Suite, SIMD, Präprozessor, Code-Transformation

1 Einleitung

Der wachsende internationale Konkurrenzdruck stellt die europäische Luftfahrtindustrie vor große Herausforderungen. Als Antwort auf die verschärften Umweltschutzanforderungen strebt sie in der Agenda 2020 an, den durch den Luftverkehr verursachten Lärm zu halbieren, sowie den Kohlendioxid-Ausstoß (und damit den Kerosinverbrauch) um 50 Prozent und die Produktion von Stickoxiden um 80 Prozent zu reduzieren. Gleichzeitig soll die Verkürzung der Entwicklungszeiten um 50 Prozent die Konkurrenzfähigkeit signifikant verbessern. Diese Ziele lassen sich nur durch konsequente Weiterentwicklung der Simulationstechnologie erreichen. Sie erfordern insbesondere den Übergang von der Simulation einzelner physikalischer Phänomene für Einzelkomponenten zu einer multidisziplinären Simulation kompletter Systeme unter realen und instationären Einsatzbedingungen. Die dazu erforderlichen Entwicklungen erfolgen in Europa, und insbesondere in Deutschland, in enger Partnerschaft

von Industrie und Großforschung.

Eine wichtige Rolle spielen hierbei große Simulationscodes, die in öffentlichen Forschungsinstituten unter Verwendung innovativer Algorithmen entwickelt werden und dadurch einen Vorsprung gegenüber kommerziell verfügbaren Produkten aufweisen. Neben dem bei der französischen ONERA entwickelten blockstrukturierten Strömungslöser elsA sind die im DLR entwickelten Codes TAU (Berechnung von Außenströmungen) und TRACE (Reagierende Innenströmungen) die tragenden Säulen der Simulationstechnik in der europäischen Luftfahrtindustrie. Die initiierte Kopplung der Codes TAU und TRACE wird in Zukunft die Simulation noch komplexerer (und damit realistischerer) Strömungsszenarien erlauben. Auf der Hardwareseite genügen für Auslegungsrechnungen i. d. R. mittelgroße Rechnerplattformen mit mehreren Hundert Rechenknoten, während Proof-of-Concept-Studien den Einsatz der jeweils stärksten installierten Rechnerplattformen (beispielsweise die Systeme der Gauß-Allianz) erfordern. Alle diese Systeme basieren bereits heute auf Multi-Core-Technologie. Für die nahe Zukunft ist der Wechsel zu Many-Core-Architekturen absehbar.

2 SIMD-Erweiterung Vampir-Suite

Um herauszufinden, in welchen Teilen die strömungsmechanischen Programme in Richtung SIMD verbessert werden können, soll die Vampir-Suite in diese Richtung erweitert werden. Bei der Vampir-Suite handelt es sich um eine Leistungsanalyse-Software, die Ereignisse (z. B. Funktionsaufrufe) von parallelen Programmen zunächst aufzeichnet (*Tracing*) und diese dann durch *globale* (für alle Prozesse) und *lokale Anzeigen* (für einen bestimmten Prozess) über die Zeit grafisch darstellt (*Visualisierung*) [1]. Abb. 1 zeigt eine typische grafische Anzeige der Vampir-Suite. Im Detail sollen die Tracing- und Visualisierung-Fähigkeiten der Vampir-Suite dahingehend verbessert werden, dass SIMD-Operationen wie z. B. Vektoradditionen oder -multiplikationen über neue grafische Anzeigen besser beobachtet werden können.

Bevor Informationen über die Verwendung von *SIMD-Maschinenbefehlen* (Maschinenbefehlen, die mehrere Daten gleichzeitig verarbeiten können) grafisch dargestellt werden können, müssen diese Informationen in einer geeigneten Form während eines Programmlaufs aufgezeichnet werden. Moderne Mikroprozessoren besitzen spezielle Register, in denen Informationen über verschiedene Ereignisse (z. B. die Anzahl der Gleitkommazahloperationen oder die Anzahl der Cache-Misses) erfasst werden können, auf vielen Prozessoren auch die Anzahl bestimmter SIMD-Maschinenbefehle.

Definition 1 (Hardware-Counter [2]). *Register auf einem Mikroprozessor, die speziell dafür bereit stehen, die Häufigkeiten des Auftretens bestimmter Hardware-Ereignisse oder Hardware-Informationen zu speichern, heißen auch Hardware-Counter.*

Definition 2 (Spezielle Hardware-Ereignisse [2]). *Ein Hardware-Ereignis, das sich (nur) mit Hilfe von Hardware-Countern des zugrundeliegenden Mikroprozessors aufzeichnen lässt, heißt natives Hardware-Ereignis. Ein natives Hardware-Ereignis, das sich aufgrund der Verwendung einer bestimmten Bibliothek unter einem weiterem Namen ansprechen lässt, heißt (von der Bibliothek) voreingestelltes Hardware-Ereignis. Ein Hardware-Ereignis, das aus anderen Hardware-Ereignissen zusammengesetzt ist, heißt abgeleitetes Hardware-Ereignis.*

Die Tabellen 1, 2 und 3 enthalten Beispiele von nativen, voreingestellten und abgeleiteten Hardware-Ereignissen für zwei verschiedene Prozessoren.

Tabelle 1. SIMD-relevante, native Hardware-Ereignisse, die auf Mikroprozessoren des Typs AMD[®] Opteron[®] X85 verfügbar sind.

Nativer Hardware-Ereignisname	Beschreibung
FR_X86_INS	Anzahl aller Maschinenbefehle
FR_FPU_MMX_3D	Anzahl SIMD-Befehle der Erweiterungen MMX [™] und 3DNow! [™]
FR_FPU_SSE_SSE2_PACKED	Anzahl gepackter SIMD-Befehle der Erweiterungen SSE und SSE 2
FR_FPU_SSE_SSE2_SCALAR	Anzahl skalarer SIMD-Befehle der Erweiterungen SSE und SSE 2
FR_FPU_X87_SSE_SSE2_SCALAR	Anzahl x87-Maschinenbefehle und skalarer SIMD-Befehle (SSE, SSE 2)
FR_FPU_X87_SSE_SSE2_SCALAR_PACKED	Anzahl x87-Maschinenbefehle und aller SIMD-Befehle (SSE, SSE 2)

Tabelle 2. Hardware-Counter-spezifische Merkmale von Mikroprozessoren des Typs AMD[®] Opteron[®] X85 sowie des Mikroprozessors Intel[®] Itanium[®] 2 Montecito 9040 bei Verwendung der Bibliothek PAPI in der Version 3.5.0.

Merkmal	Opteron [®] X85	Itanium [®] 2 Montecito 9040
Anzahl Hardware-Counter	4	12
Native Hardware-Ereignisse	248	637
Von PAPI voreingestellte Ereignisse	103	103
davon verfügbar	42	60
davon verfügbar und abgeleitet	13	42

Um in Programmierhochsprachen auf Informationen aus Hardware-Countern zugreifen zu können, werden üblicherweise spezielle Bibliotheken verwendet, die solche Informationen in Form einer API zur Verfügung stellen. Das Tracing-Werkzeug der Vampir-Suite unterstützt die Verwendung der Bibliothek *PAPI* („Performance Application Programming Interface“) [3], [4]. Diese stellt Hardware-Informationen in C und C++ zur Verfügung und enthält in der Version 3.5.0 ca. 100 voreingestellte Hardware-Ereignisse. Damit ist der Zugriff auf Hardware-Counter aus der Vampir-Suite heraus möglich, und es können dann z. B. für verschiedene numerische Benchmarks SIMD-relevante Hardware-Ereignisse aufgezeichnet und visualisiert werden. Ein Beispiel für solche Visualisierungen ist in Abb. 2 gegeben.

3 Skalierbare Algorithmen für Many-Core-Architekturen

In diesem Teilarbeitspaket soll untersucht werden, wie sich die parallele Effizienz von Strömungslösern auf Multiblocknetzen weiter erhöhen lässt. Die Entwicklung eines generischen Many-Core-fähigen Partitionierungswerkzeugs für blockstrukturierte Gitter ist hier unerlässlich. Das Ziel ist zum einen die Entwicklung effektiver Verfahren zur Verteilung sowie zum Splitten und Mergen von Teilblöcken zu einem optimalen Verbund in Abhängigkeit der zur Verfügung stehenden Prozessoren.

Dabei sollen zusätzlich Anforderungen bezüglich Lastbalancierung, Cache-Nutzung, Kaskadierung des Kommunikationsnetzwerks auf parallelen Many-Core-Systemen sowie physikalische und softwaretechnische Nebenbedingungen berücksichtigt werden. Der Partitionierer soll nach Ende des Projekts frei verfügbar gemacht werden.

Folgendes Vorgehen ist geplant:

Tabelle 3. SIMD-relevante, von der Bibliothek PAPI (Version 3.5.0) voreingestellte Hardware-Ereignisse, die auf Mikroprozessoren des Typs AMD[®] Opteron[®] X85 verfügbar sind.

Voreingestellter Ereignisname	Nativer Hardware-Ereignisname
PAPI_FP_INS	FR_FPU_X87_SSE_SSE2_SCALAR_PACKED
PAPI_VEC_INS	FR_FPU_SSE_SSE2_PACKED
PAPI_TOT_INS	FR_X86_INS

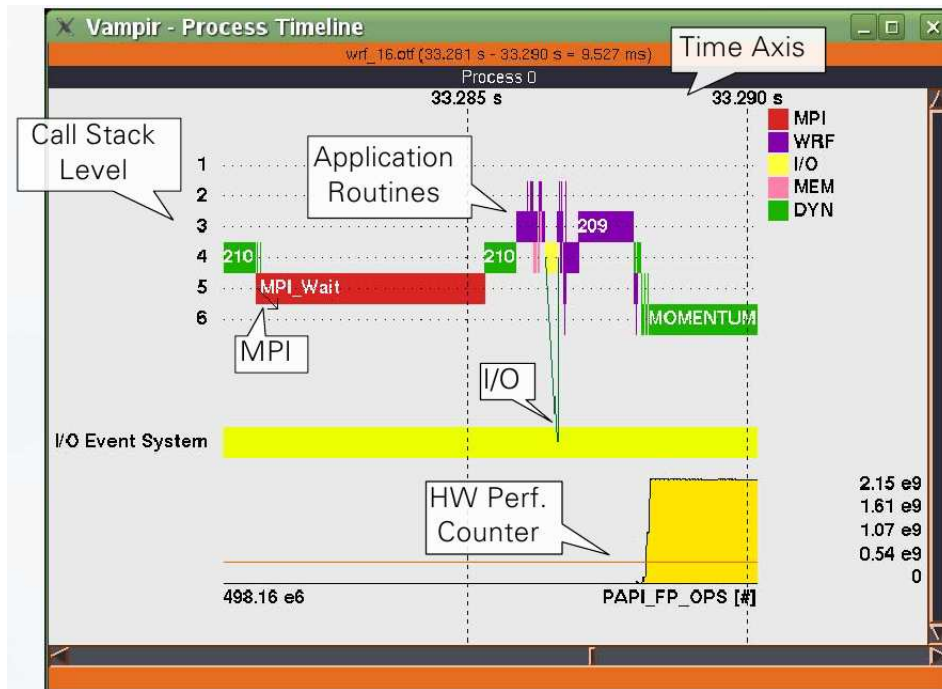


Abbildung 1. Typische Anzeige innerhalb der Leistungsanalyse-Werkzeuges Vampir zur Untersuchung des Leistungsvermögens eines parallelen Programms.

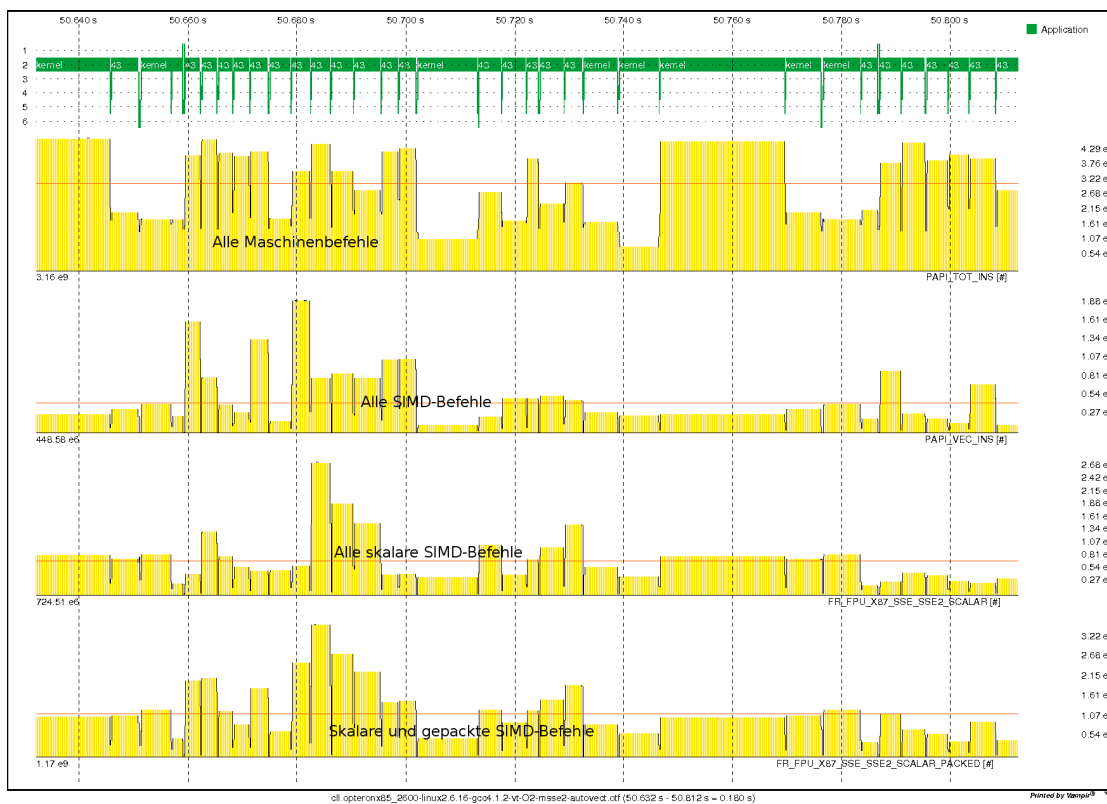


Abbildung 2. Grafische Darstellung verschiedener Hardware-Counter über den Programmverlauf eines numerischen Benchmarks.

- Zunächst wird eine Kostenfunktion für die Rechenkosten pro Block definiert. Diese berücksichtigt unter anderem Löseroperationen, insbesondere Vorkonditionierung und Rechnungen bezüglich Randbedingungen, z.B. periodischen Randbedingungen.
- Gemäß der Kostenfunktion sehr teure Blöcke werden dann gesplittet, um später eine last-balancierte Verteilung der Blöcke auf die einzelnen (Multi-Core-)Prozessoren erreichen zu können.
- Anschließend wird ein „Block“-Graph erstellt. Die Knotengewichte dieses Graphen entsprechen den Rechenkosten pro Block, die Kantengewichte den Kommunikationskosten zwischen den Blöcken.
- Die „Block“-Graphpartitionierung auf die einzelnen Prozessoren kann dann z.B. mit ParMETIS [5] durchgeführt werden.
- Während sich die Graphpartitionierung auf die MPI-Parallelisierung bezieht, können die Berechnungen pro Multi-Core-Prozessor per OpenMP-Parallelisierung weiter auf die einzelnen Cores verteilt werden. Die Datenaufteilung muss hier die Speicherhierarchie des Prozessors berücksichtigen.

Zum anderen muss der Löser für lineare Gleichungssysteme für Multi-Core-Architekturen optimiert und die parallele Skalierbarkeit des Lösungsalgorithmus verbessert werden. Dazu sollen verschiedene Vorkonditionierer in TRACE implementiert und auf ihre Performance untersucht werden.

In einem ersten Test wurde die in *linear-/adjointTRACE* vorhandene Block-Jacobi-Vorkonditionierung (BJ) [6] mit der „Distributed Schur Complement“-Methode (DSC) [7] anhand eines typischen Matrix-Problems verglichen. Die BJ-Methode berücksichtigt lediglich die lokalen Diagonalblöcke der verteilten Matrix und vernachlässigt Kopplungen zwischen den Teilgebieten. Daher verschlechtert sich der Vorkonditionierungseffekt mit steigender Prozessorzahl. Andererseits nimmt der Aufwand zur Berechnung des Vorkonditionierers ab. Die DSC-Methode hingegen berücksichtigt alle Kopplungen zwischen den Teilgebieten. Der Aufwand zur Bestimmung und Anwendung des Vorkonditionierers nimmt jedoch mit wachsender Prozessorzahl in der Regel zu, da die Zahl der Kopplungen in der Regel ansteigt.

Abbildung 3 vergleicht die Ausführungszeiten eines parallelen FGMRes-Lösers [6] mit BJ- bzw. DSC-Vorkonditionierung anhand eines typischen, kleinen *linearTRACE*-Matrix-Problems (Matrixordnung: 56240; Nichtnullelemente: 2572040; Kondition: $8,4 \cdot 10^6$). Die Performance-Untersuchung wurde auf 1 bis 64 Prozessoren eines Cluster von DLR-AT durchgeführt (AMD Opteron Processor 250; Dual-Processor Nodes; 2,4 GHz).

Abbildung 3 (links) zeigt, dass bei kleinen Prozessorzahlen das unaufwendigste BJ-Verfahren (unvollständige *LU*-Zerlegung der Diagonalblöcke mit Threshold 10^{-2} [6]) die günstigsten Ausführungszeiten liefert. Ab 16 Prozessoren wird das etwas teure BJ-Verfahren mit Threshold 10^{-3} gleichwertig und erzielt ab 32 Prozessoren ein besseres Zeitverhalten. Ab 32 Prozessoren lohnt sich die aufwendigere DSC-Methode mit dem besten Parametersatz (Threshold 10^{-3} , 5 innere Iterationen für das Kopplungssystem [8]) und liefert ab 64 Prozessoren ca. 25% kürzere Ausführungszeiten als die beste BJ-Methode (Abbildung 3, rechts).

Die Performance-Untersuchung lässt den Schluss zu, dass sich die aufwendige DSC-Methode gegenüber dem einfachen BJ-Verfahren bei hohen Prozessorzahlen für *linearTRACE*-Probleme lohnt. Wünschenswert ist ein intelligenter Löser mit Problem- bzw. konvergenzabhängiger Parametersteuerung und Problem- bzw. konvergenzabhängiger Vorkonditionierung (z.B. BJ bei kleinen Prozessorzahlen und DSC bei hohen Prozessorzahlen).

Für den auf unstrukturierten Netzen arbeitenden Strömungslöser TAU soll ferner untersucht werden, ob sich die parallele Effizienz des verwendeten Mehrgitterverfahrens weiter erhöhen lässt.

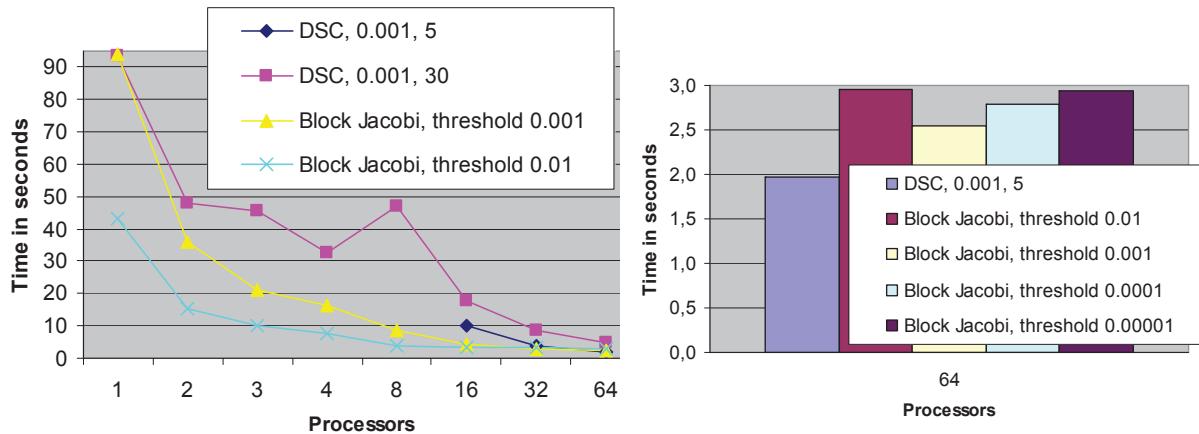


Abbildung 3. Performance-Vergleich eines parallelen FGMRes-Lösers mit BJ- bzw. DSC-Vorkonditionierung.

4 Hybride OpenMP/MPI-Implementierung TAU/TRACE

Beim TAU-Strömungslöser wird eine mehrstufige SPMD-Gebietszerlegung implementiert (SPMD/MPI – SPMD/OpenMP). Die MPI-Domänen selbst werden dabei vollständig in Sub-Domänen zerlegt. Diese Sub-Domänen sind dabei auf die Größen der entsprechenden L2- bzw. L3-Caches angepasst. Die entstehende hohe zeitliche und räumliche Datenlokalität erlaubt eine hocheffiziente Nutzung der Caches. Mit Hilfe der SPMD/OpenMP-Implementierung können die Sub-Domänen parallel abgearbeitet werden. Durch die zweistufige MPI/OpenMP-Parallelisierung kann eine extrem hohe parallele Effizienz erreicht werden. Insbesondere ist der Synchronisationsaufwand zwischen den OpenMP-Threads auf den Cores relativ gering. Zusätzlich muss in diesem Schritt auch das Zusammenspiel zwischen dem SIMD-Präprozessor und den numerischen Kernels von TAU gewährleistet werden.

Beim TRACE-Strömungslöser kommt ebenfalls eine hybride OpenMP/MPI-Implementierung zum Tragen. Durch die von TRACE verwendeten blockstrukturierten Gitter ist eine klassische, schleifenbasierte OpenMP-Lösung pro MPI-Domäne erfolgversprechend. Hierbei soll ein Analysewerkzeug auf Schleifenebene geschaffen werden, das die notwendigen Direktiven unter Berücksichtigung von shared und private Variablen automatisch einfügt und die Schleifen restrukturieren und optimieren kann. Somit steht dem Modellierer bei der Erweiterung des Simulationsprogramms die hybride Parallelisierung direkt zur Verfügung.

In Analogie zum TAU-Strömungslöser muss hier ebenfalls das Zusammenspiel zwischen dem SIMD-Präprozessor und den numerischen Kernels von TRACE gewährleistet werden.

Basierend auf der Erfahrung beim Umgang mit den numerischen Kernels der beiden Strömungslöser werden beide Applikationen mit entsprechenden Direktiven instrumentiert.

5 Generischer SIMD-Präprozessor

Basierend auf den Anforderungen der extrahierten numerischen Kerne der beiden Strömungslöser TAU und TRACE soll ein generischer SIMD-Präprozessor entwickelt werden.

Das Ziel ist dabei die Entwicklung eines Programmierwerkzeuges, das einerseits automatisch den Quellcode eines Programms transformiert, andererseits dem Benutzer über eine grafische Oberfläche erlaubt, gezielt selbst Quellcodebereiche zur Transformation auszuwählen (siehe Abb. 4). Damit wird dem Entwickler sowohl bei der SIMD-Optimierung als auch bei der Optimierung des Cache-Verhaltens die fehleranfällige und langwierige Arbeit der Code-Änderung abgenommen.

Durch Direktiven soll der Entwickler Quellcodetransformationen anweisen können, die der Präprozessor umsetzt und den so generierten neuen Quellcode dem Entwickler zur Kontrolle präsentiert. Weiterhin sollen auf Anweisung hin automatisch Instrumentierungsroutinen für die Vampir-Suite in den Quellcode eingefügt werden, um mit Hilfe der erweiterten Tracing-Möglichkeiten der Vampir-Suite (siehe Abschnitt 2) entstandene Änderungen überprüfen zu können. Die Implementierung des SIMD-Präprozessors soll auf Vorarbeiten eines Präprozessors zur Cache-Optimierung aufsetzen, welcher im Rahmen eines Forschungsprojektes an der TU Dresden entwickelt wurde [9].

Der Präprozessor soll in zwei Stufen implementiert werden. In der ersten Stufe werden die folgenden allgemeinen Code- und Schleifentransformationen implementiert:

1. *Inlining*: (Beim Inlining wird innerhalb des Quellcodes der Aufruf einer Funktion durch den Funktionskörper der aufrufenden Funktion explizit ersetzt.) Damit überhaupt weitere Quellcodetransformationen möglich sind, sollen Funktionsaufrufe in der innersten Schleife mittels Inlining ersetzt werden. Der Programmierer muss dabei dafür sorgen, dass die Parameterliste keine Alias-Argumente enthält.
2. *Schleifeninvariante/indexvariante Verzweigung*: (Eine Verzweigung in einer Schleife, bei der das Ergebnis von Daten des aktuellen Schleifendurchlaufs unabhängig ist, heißt schleifeninvariante Verzweigung, eine Verzweigung innerhalb einer Schleife, bei der das Ergebnis nur vom Schleifenindex abhängig ist, heißt indexvariante Verzweigung). Bei einer schleifeninvarianten Verzweigung soll der Schleifenkörper zerlegt werden, bei einer indexvarianten Verzweigung sollen beide Möglichkeiten gerechnet und danach ein Ergebnis verworfen werden.
3. *Loop-Unrolling*: (Ist eine Schleife mit einer festen Anzahl m an Schleifendurchläufen vorgegeben, dann wird die Schleife *abgerollt*, falls die Anweisungen im Schleifenkörper mit entsprechend angepassten Schleifenindizes n -mal ($n \leq m$) explizit wiederholt werden [10, S. 180]).
4. *Loop-Splitting*: (Ist eine Schleife gegeben, die in ihrem Schleifenkörper mehrere, voneinander unabhängige Anweisungen besitzt, dann wird die Schleife *aufgeteilt*, falls für jede Anweisung im Schleifenkörper eine eigene, neue Schleife erstellt wird [11, S. 666].) Beim Loop-Splitting sollen skalare Variablen in so genannte Workarrays überführt werden. Dabei ist zu beachten, dass diese Arrays von der Datengröße her in den L1-Cache passen. Dies lässt sich typischerweise mittels Loop-Blocking der äußeren Schleife und entsprechenden inneren Schleifen mit Längen von maximal 8 bis 16 erreichen.

Die obigen Transformationen bilden die Grundlage für die zweite Stufe. In dieser Stufe sollen dann Quellcodeteile entweder direkt durch so genannte *intrinsische SIMD-Funktionen* (spezielle Funktionen eines Compilers, die SIMD-Maschinenbefehle verwenden) ersetzt werden oder der Compiler bei ihrer automatischen Vektorisierung durch den Präprozessor unterstützt werden. Anhand von verschiedenen Benchmarks soll entschieden werden, welcher Ansatz gewählt wird.

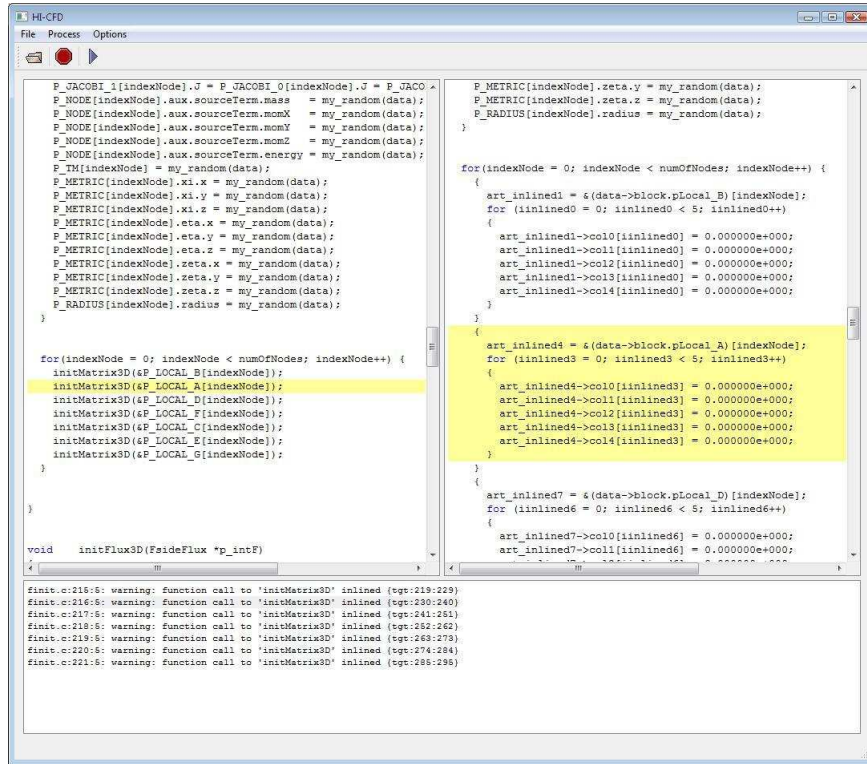


Abbildung 4. Schnappschuss der grafischen Oberfläche des SIMD-Präprozessors.

6 Evaluierung TAU/TRACE

Der erfolgreiche Einsatz der Strömungslöser in Forschung und Industrie hängt nachhaltig vom Qualitätsmanagement der Software ab. Hierbei seien die Aspekte Konsistenz der Lösungen zu früheren Versionen, numerische Stabilität als auch Wartbarkeit für die vielfältige weitere Entwicklung auf verschiedenen Modellebenen genannt. Dies ist Voraussetzung, damit die in diesem Projekt geänderten Strömungslöser eine Basis für alle zukünftigen Entwicklungen bilden können.

Eine Evaluierung an industrierelevanten Testfällen soll die erhöhte Wettbewerbsfähigkeit der in HICFD modifizierten Simulationscodes demonstrieren.

Es wird ein Best-Practice-Report erstellt, der die Themengebiete „Hybride Parallelisierung“ und „SIMD-Nutzung“ behandeln wird. Hierin werden Vorgehensweisen, Hinweise und Erfahrungen aus diesem Projekt dem wissenschaftlichen Umfeld zur Verfügung gestellt. Der Best-Practice-Report soll ferner als Anleitung dienen, wie Simulationscodes auch aus anderen Disziplinen als der Strömungsmechanik für die effiziente Ausführung auf Many-Core-Architekturen optimiert werden können.

Danksagung. Das Projekt wird vom Bundesministerium für Bildung und Forschung unter den Kennzeichen IH08012A, B, C im Rahmen des Programms „IKT 2020 – Forschung und Innovation“ gefördert.

Literatur

1. Hartmut Mix und Anika Uhlemann: VampirServer 1.09 User Manual. GWT-TUD, Dresden. (2008)
2. B. Sprunt: The Basics of Performance-monitoring Hardware. IEEE Micro **22**(4) (2002) 64–71

3. Dan Terpstra: PAPI Users Guide Version 3.5.0. Innovative Computing Laboratory, University of Tennessee Department of Electrical Engineering and Computer Science, Knoxville. (2006)
4. TU Dresden, ZIH: VampirTrace 5.6 User Manual. TU Dresden, ZIH, Dresden. (2008)
5. G. Karypis und V. Kumar: ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. University of Minnesota, Minneapolis. (1997)
6. Yousef Saad: Iterative Methods for Sparse Linear Systems (Second Edition). SIAM, Philadelphia (2003)
7. Y. Saad und M. Sosonkina: Distributed Schur Complement Techniques for General Sparse Linear Systems. *SISC* **21** (1999) 1337–1356
8. A. Basermann, U. Jaekel, M. Nordhausen und K. Hachiya: Parallel Iterative Solvers for Sparse Linear Systems in Circuit Simulation. *Future Generation Computer Systems* **21** (2005) 1275–1284
9. R. Müller-Pfefferkorn, W. E. Nagel und B. Trenkler: Optimizing Cache Access: A Tool for Source-To-Source Transformations and Real-Life Compiler Tests. M. Danelutto, M. Vanneschi, D. Laforenza (Eds): Euro-Par 2004, Parallel Processing **LNCS 3149** (2004) 72–81
10. Wolfram Schiffmann und Robert Schmitz: Technische Informatik 2: Grundlagen der Computertechnik. Springer-Verlag (2005)
11. Randy Allen und Ken Kennedy: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. NetLibrary (2001)

Global Cellular Automata: a Path from Parallel Random Access Machines To Practical Implementations^{*}

Jörg Keller¹ and Andre Osterloh^{2**}

¹ FernUniv. in Hagen, Dept. of Math. and Computer Science, 58084 Hagen, Germany
joerg.keller@fernuni-hagen.de

² Techn. Univ. Darmstadt, Dept. of Computer Science, 65289 Darmstadt, Germany
osterloh@andre-osterloh.de

Abstract. The Parallel Random Access Machine (PRAM) and the Global Cellular Automaton (GCA) are both models to study parallel algorithms. We investigate the commonalities and differences between these models, because we think that they could nicely complement each other: PRAMs have lots of algorithms, GCAs provide efficient implementation paths. We provide the following results: GCAs are CROW PRAMs, GCAs have more advantageous optimality criteria than PRAMs, and GCA implementations are more efficient than typical PRAM emulations.

Key words: PRAM simulation, Global Cellular Automaton, Optimality Criteria

1 Introduction

The Parallel Random Access Machine (PRAM) is an established model to study parallel algorithms without consideration of details such as interconnection network or synchronization cost. A number of processors work synchronously on a shared memory with unit access time. There are several variants that differ on the access patterns that are allowed, denoted by $XRYW$, where $X, Y \in \{C, E, O\}$ denote the variant for read (R) and write (W) access, respectively. C means concurrent, i.e. several processors may access the same memory cell in one step. E means exclusive, i.e. only one processor may access a cell in each step. O means owner, i.e. only the owner of a cell may access this cell. While many PRAM algorithms have been developed, and several schemes have been derived to emulate a PRAM on a message-passing machine including hardware prototypes, PRAM algorithms have not found their way yet into practical parallel programming. This hurts more and more as parallel programming is becoming commonplace with multicore processors being the standard for PCs and notebooks. For details on PRAMs and their emulation, we refer to [1,2,3,4].

^{*} Research supported by German Science Foundation (DFG) grant HO 893/7-1.

^{**} Now with BTC AG, Escherweg 5, 26121 Oldenburg, Germany.

The global cellular automaton (GCA) [5] is an extension of the classical cellular automaton (CA) model. In the GCA, a collection of finite state automata (FSA) called nodes or cells update their local states synchronously where each FSA uses as input its own current state and the current state of its neighbor(s). While the neighbors are fixed by some geometrical rule in the CA, neighbors in the GCA are not fixed by physical interconnect but neighbors are identified by addresses which are part of the local state and thus can vary with node ID or over time. The number of addressable neighbors per node is a parameter of a particular GCA, and typically denoted as the number of hands, i.e. in a one-handed GCA, each node can only address one neighbor. A number of techniques to implement GCAs in reconfigurable hardware (FPGA) or non-configurable hardware (ASIC) have been developed: e.g. fully parallel [6] or data-parallel [7]. A programming and synthesis environment is available [8], i.e. there is a high-level programming language to formulate GCA algorithms, a simulator to debug them, and there exists a compiler producing VHDL code that can be synthesized into an FPGA configuration by a silicon compiler. As the reconfigurability together with programmability allows highly adapted and optimized implementations of parallel algorithms, the GCA seems a promising candidate for a highly efficient massively parallel computing platform. Yet, the number of algorithms developed for the GCA model is still rather small.

In the present paper, we investigate the commonalities and differences between both models, because they could nicely complement each other: PRAMs have lots of algorithms, GCAs provide mature and efficient implementation paths. We give our results in three parts. First, we compare GCAs and PRAMs and show how a PRAM can be simulated by a GCA. As a consequence the wealth of PRAM algorithms is accessible to GCAs. One further result of this comparison is that there is a one-to-one correspondence between a one-handed GCA and a CROW-PRAM, i.e. a GCA can be seen as a CROW PRAM. Second, we show that GCAs have more advantageous optimality criteria than PRAMs, which allows to use simpler PRAM algorithms without penalty. Third, we show that GCA implementations are more efficient than typical PRAM emulations. The similarities between CROW PRAM and GCA and aspects of optimality criteria have been mentioned rather informally in [6].

The remainder of this paper is organized as follows. In Sect. 2 we describe how one step of a PRAM can be simulated on a GCA and give upper and lower bounds. In Sect. 3 we discuss optimality criteria for PRAMs and GCAs. In Sect. 4 we describe optimal PRAM emulations via GCAs. Finally, in Sect. 5 we give a conclusion.

2 Correspondence of PRAMs and GCAs

In this section we give lower and upper bounds for the number of steps necessary to simulate t steps of a PRAM on a GCA. A detailed description of the GCA model can be found in [5]. For a description of the PRAM model see [1]. For the

rest of this section we assume the instruction sets of the PRAM and the GCA to be equal.

First, we describe a simulation of one step of a priority CRCW PRAM on a GCA. In contrast to a PRAM, a GCA does not have a common memory. Hence, to simulate a PRAM by a GCA we have to simulate the access to the memory. To distinguish between PRAMs and GCAs we call the storage locations of a GCA cells. Note that cells are able to execute instructions. Since all GCAs, even the 1-handed ones, are able to read from all their cells simultaneously, a simulation of reading access to the PRAM memory is no problem. Hence we focus our attention to simulate writing access. Here it is important to remember that a cell is not allowed to write on other cells.

We begin with the description of a simple simulation of one step of a PRAM. Then we improve the simulation to our final result. We simulate a step of a priority CRCW PRAM with p processors and m storage locations. In a priority CRCW PRAM writing conflicts are solved by using priorities. If two or more processors want to write into the same storage location the processor with the highest priority wins. We assume for simplicity and without loss of generality that the priority of a processor is given by its number, i.e. the processor with the highest number has the highest priority. We simulate the PRAM by a 1-handed GCA with $p \cdot m$ cells $c_{i,j}$, $1 \leq i \leq p$, $1 \leq j \leq m$. The basics behind the simulation are:

- (1) cell $c_{i,1}$, $1 \leq i \leq p$, simulates processor i
- (2) cell $c_{1,i}$, $1 \leq i \leq m$, simulates storage location i
- (3) cells $c_{1,i}, \dots, c_{p,i}$, $1 \leq i \leq m$, are used to solve writing conflicts on storage location i

This situation is depicted in Fig. 1.

Writing on a storage location i , $1 \leq i \leq m$, is simulated by the following three steps:

- (1) For all j , $1 \leq j \leq p$: If processor j wants to write on storage location i , the cell $c_{j,i}$ stores value i otherwise it stores value 0.
- (2) Cells $c_{1,i}, \dots, c_{p,i}$ calculate which processor writes on i . Since the processor with the highest number has the highest priority, this can be done by calculating the maximum of the stored values in $c_{1,i}, \dots, c_{p,i}$. Assume that k , $1 \leq k \leq p$, is the calculated maximum. Then k is stored in $c_{1,i}$.
- (3) $c_{1,i}$ reads the value processor k wants to write into storage location i .

In step (1) cell $c_{j,i}$ reads on cell $c_{j,1}$. This can be done in $O(1)$ steps. In step (2) a maximum of p values is calculated. This can be done in $O(\log p)$ steps [9]. So we get a first result

Lemma 1. *One step of a p -processor priority CRCW PRAM with m storage locations can be simulated by a 1-handed GCA with $p \cdot m$ cells in $O(\log p)$ steps.*

One is able to reduce the resources from $p \cdot m$ cells, each containing a polylogarithmic number of bits, to $p + m$ cells containing $p \cdot m$ bits in total. To do this

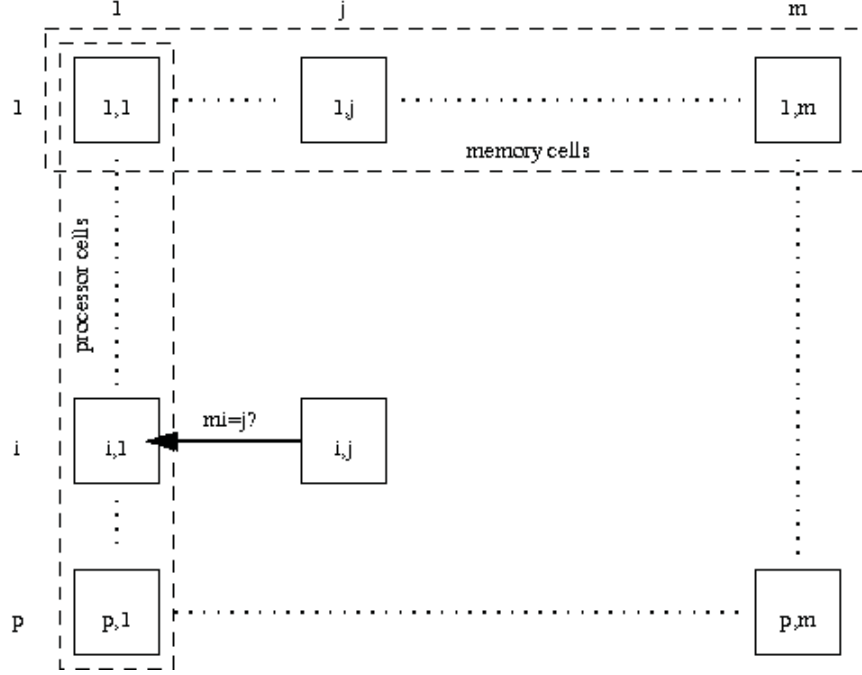


Fig. 1. Simple GCA to simulate concurrent writes.

we have to assume that all processor cells of the GCA are able to store at least m bits. The idea is the following: we take a balanced binary tree of cells. We call the root of the tree r . In this tree the p cells simulating the processors are the leaves. Such a tree consists of $2p - 1$ cells. Each cell of the tree is equipped with a register of m bits. Additionally m other cells exist. These cells simulate the m storage locations. If processor i wants to write on storage location j , the leaf with number i sets the j -th bit of its register. Using a standard technique we calculate in $O(\log p)$ steps in the register of r on which memory locations we want to write, i.e. the i -th bit in the register of r is set if and only if one processor cell wants to write to storage location i . This situation is depicted in Fig. 2. Now each of the m cells is able to calculate in one step by testing the register in cell r if it shall be written. If so, it can calculate in $O(\log p)$ steps which processor cell wants to write to it, by following the path from root r to the appropriate leaf processor cell, which is a standard tree search technique. The memory cell then reads the value to be written from that processor cell, and stores it locally. Hence we get:

Theorem 1. *One step of a p -processor priority CRCW PRAM with m storage locations can be simulated in $O(\log p)$ steps of a 1-handed GCA with $O(p + m)$ cells holding $O(p \cdot m)$ bits in total.*

We are also able to reduce the number of cells from $p \cdot m$ to $O(p + m)$, each containing only a polylogarithmic number of bits, by paying with some extra simulation time. To do this, we employ a GCA with p cells representing the PRAM processors and m cells representing the memory locations. In the PRAM

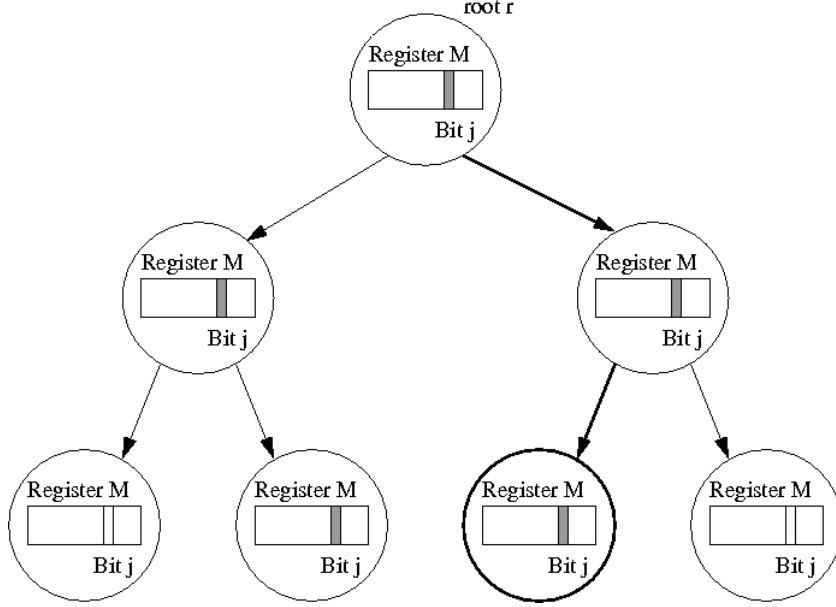


Fig. 2. Tree of GCA cells for concurrent write.

step to be simulated, each processor P_i that wants to write value v_i to shared memory location a_i generates a tuple

$$(a_i, i, v_i) ,$$

while other processors choose $a_i = \infty$. The processors sort these tuples, which can be achieved with the help of bitonic sort in $O((\log p)^2)$ GCA steps. To handle concurrent writes, each processor P_i compares its local tuple to the tuple in P_{i+1} . If both tuples contain the same address, then processor P_i changes its local address to ∞ , so that only one write to each memory location remains, originating from the processor with highest ID, i.e. highest priority. Now the processors sort these tuples again. Then, each memory cell can apply a binary search among the tuples to check if its address is present. If so, the memory cell performs the write. This takes $O(\log p)$ GCA steps. Thus we arrive at the following theorem.

Theorem 2. *One step of a p -processor priority CRCW PRAM with m storage locations can be simulated by a 1-handed GCA with $O(p+m)$ cells in $O((\log p)^2)$ steps.*

Now we show that we generally cannot reduce the number of steps necessary to simulate one step of a PRAM below a logarithmic bound.

Theorem 3. *Let $a \in \mathbb{N}$. An a -handed GCA needs $\Omega(\log_{a+1} p)$ steps to simulate one step of a CREW PRAM with p processors. This result is independent from the instruction set and the number of cells of the CGA.*

Proof. Given is an a -handed GCA G . The task is to calculate a value that depends on p inputs. The inputs are given in cells $c_1, \dots, c_p$. The result has to be stored in cell c_{p+1} . In the following we show that G needs $\Omega(\log_{a+1} p)$ steps to calculate the result in cell c_{p+1} .

Before the calculation of the result starts, the content of c_{p+1} depends on no other cell. Since G is a -handed, the state of each cell, in particular c_{p+1} , depends at most on $a + 1$ cells (its own state and a other cells) after the first step of a calculation. After $t \geq 2$ steps of a calculation, the state of c_{p+1} depends on the cells it depended on after step $t - 1$, and additionally on the cells it accessed in step t . So the maximum number x_t of cells that c_{p+1} depends on after t steps is given by

$$x_1 = a + 1 \tag{1}$$

$$x_t = x_{t-1} + a \cdot x_{t-1}, \quad t \geq 2. \tag{2}$$

So $x_t = (a + 1)^t$. Since the input is given in cells $c_1, \dots, c_p$ and the result depends on all of them, the calculation can only be finished for a t with $(1 + a)^t \geq p$, i.e. the desired result.

Finally we shortly discuss why CROW PRAMs are GCAs. We have seen that the simulation of a read access on a storage location is no problem. Since processors of a CROW PRAM are only allowed to write on a storage location where they have owner rights and furthermore each storage location has at most one owner, no write conflicts can occur. So the storage locations for which a processor is owner can be coded in the state of the cell that simulates the processor. In order to arrive at GCA cells with asymptotically equal state sizes, one must further request that each PRAM processor is owner of $O(m/p)$ memory locations.

We also assume that in a p -processor PRAM with m memory locations, normally m is polynomial in p and each memory location contains a number of bits polylogarithmic in m .

3 Optimality Criteria in PRAMs and GCAs

While the computational power of GCAs and PRAMs is equal (see previous section), there are still some differences. In a PRAM, one counts the number of processors as cost, but does not take the memory size and cost into account. This may result from the impression that a processor is a very complex circuit comprising more than 10^9 transistors, while memory cells are incredibly cheap, involving about one transistor per bit for a dynamic RAM cell, or four transistors for a static RAM cell. In contrast, in a GCA each node has a state that comprises both processor state and memory state, and has some circuits for state transitions from cycle to cycle. Thus, counting the nodes of a GCA involves both processing and storage cost, with the ratio between cost per processor and cost per memory bit being very much smaller than in the PRAM model. As a GCA algorithm to be executed is directly encoded into the nodes' finite state machines,

a GCA node typically contains only few hundred logic gates for processing. Thus, memory makes out a considerable fraction of the node cost.

This difference becomes apparent when we consider the processor-time product (PTP) of a parallel algorithm. When using a PRAM, the PTP is the product of the parallel algorithm's runtime t_p on a PRAM and the number P of processors of the PRAM. The ratio behind this definition is the ability to compare the parallel algorithm to the sequential complexity t_s of the problem to be solved. The PTP cannot be asymptotically smaller than this sequential complexity. If we assume that we would have a parallel algorithm with a smaller PTP, then we could simulate a PRAM with this algorithm sequentially, instruction by instruction of all PRAM processors, and would arrive at a sequential algorithm with a runtime that asymptotically equals the PTP, which contradicts the above assumption. If the PTP equals the sequential complexity, i.e. if

$$t_p \cdot P = O(t_s) , \quad (3)$$

then the parallel algorithm is *PTP-optimal*. Consequently, there has been a strive in PRAM algorithmics to find PTP-optimal parallel algorithms that use as many processors as possible.

Yet, these are two conflicting goals. A parallel algorithm without concurrent write that takes input of size n into account completely has a parallel runtime of at least $t_p = \Omega(\log n)$. Because of Eq. (3), there exist only PTP-optimal parallel algorithms using at most

$$P = O(t_s/t_p) = O(t_s/\log n) \quad (4)$$

processors. As in many problems, especially the ones with linear time complexity, the memory consumption is $\Theta(t_s)$, the memory size is asymptotically larger than the processor count, but is not taken into account.

Creation of a PTP-optimal algorithm often occurs in two steps: there is a parallel algorithm that achieves optimal parallel runtime, but needs more processors than given in Eq. (4), typically on the order of the sequential complexity, i.e. $P = O(t_s)$. Then one modifies this algorithm, for example by reducing the problem size by a factor of $\log n$ in a preprocessing step, and then one applies the non-optimal algorithm on the reduced problem. As $\log(n/\log n) = \Theta(\log n)$, the algorithm now is PTP-optimal. Yet, the construction of a PTP-optimal algorithm may be much more elaborated, so that it would be advantageous to be able to restrict to the non-optimal algorithms which are much simpler. Also, in this case, memory size and processor count would be proportional.

As an example, consider the problem of computing the sum of n numbers. A very simple PRAM algorithm with $P = n/2$ processors repeatedly adds two of the numbers with each processor, until the sum remains, see the following pseudocode.

```

int numbers[n];
int sum;

simplesum(int n){ // proc. count equals array size
  for(t=log(n)-1;t>=0;t--){
    if(PID<(1<<t)){ // use 2^t proc.s in step t
      numbers[PID] = numbers[2*PID] + numbers[2*PID+1];
    }
  }
  sum = numbers[0];
}

```

This simple algorithm uses $P = n/2$ processors to sum n numbers in time $t_p = O(\log n)$. Thus it is time-optimal as the sequential complexity is $t_s = O(n)$, but not PTP-optimal. If we can reduce our problem size to $n/\log n$ in parallel time $O(\log n)$ with $n/\log n$ processors, then the simple algorithm can be applied with $n/\log n$ processors in time $O(\log(n/\log n)) = O(\log n)$. The resulting algorithm is PTP-optimal. Such a preprocessing is easily achieved by having each of the $n/\log n$ processors simply sum up $\log n$ numbers, see the following pseudocode.

```

int numbers[n];
int sum;

optsum(int n){ // proc. count is only n/log n
  for(i=1;i<log(n);i++) numbers[PID] += numbers[PID+i*n/log(n)];
  simplesum(n/log(n));
}

```

In contrast, while there exists a simple time-optimal PRAM algorithm for ranking a list of size n with n processors [10], reducing the number of processors to $n/\log n$ requires complex programming means like independent set removal [11].

Another argument why the PTP as defined above might not give best results in reality is a re-interpretation of the PTP. If we see runtime as the inverse of the performance (performance is defined as work per time, and for a given algorithm the work to be done is fixed), then the PTP is also a measure of price per performance. However, this price or cost only includes processors but not the memory cost.

In contrast to the PRAM, the node cost in a GCA includes memory cost as the memory cells are part of the node states. Hence there is no necessity to reduce the processor count below the memory size, i.e. at least the problem size, because this would not simplify the GCA. In turn, this relaxes the PTP-criteria.

Thus, our second result is that in the GCA, optimality criteria are relaxed, and thus non-PTP-optimal PRAM algorithms become optimal in the GCA. Non-PTP-optimal but time-optimal PRAM algorithms typically exist and are much simpler than PTP-optimal PRAM algorithms. As the GCA can avoid these

elaborate schemes, simpler algorithms can be applied without any penalty which increases productivity. Also, these simpler algorithms further help the GCA by simplifying the state machines and thus reduce node cost and potentially allow to increase clock frequency.

4 Optimal PRAM emulations via GCAs

While GCAs seem to relate favorably to PRAMs, the realization of reads to arbitrary addresses, possibly concurrent reads, takes time also in GCA implementations. Hence, the question arises whether GCAs only provide new and better insights into optimality of algorithms, or whether they also lead to more efficient implementations than direct PRAM emulations on processor networks, which have been studied for decades.

Emulations of a given P -processor PRAM on a parallel message-passing machine (MPP) typically work in the following manner. The PRAM address space is distributed over the MPP memories in a pseudo-random manner. The size p of the MPP is chosen such that $P = l \cdot p$ and that an l -relation³ can be handled by the MPP's interconnection network in time $O(l)$. For example, this is true for a coated mesh of size $p \times p$, where p processors occupy only the first and the last row of the mesh, and the routing is done in time $l = O(p)$ [12]. This also holds probabilistically for a wrapped butterfly network of p rows, with processors only present in the first column of the butterfly network. This situation is depicted in Fig. 3(a). The routing of an $l = O(\log p)$ relation takes time $O(\log p)$ in a butterfly network with high probability [13].

Each MPP-processor simulates l PRAM processors, instruction by instruction. Accesses to shared addresses are realized via requests (and responses in case of reads) over the interconnection network to the MPP-memories holding those addresses. Concurrent accesses to some address are handled by combining the requests in the interconnection network or at the memory module holding this address, in order to avoid a hot spot [14,15].

Thus one step of P PRAM processors can be simulated in time $\Theta(l)$ on a p -processor MPP. The Θ is necessary because the interconnect latency gives an upper bound, but the simulation of one step of l PRAM processors on one MPP processors still takes time $\Omega(l)$. As $P = l \cdot p$, this is time-optimal. As the latency in constant-degree networks is $\Omega(\log p)$ or even $\Omega(\sqrt{p})$ if we restrict to constant length wiring between nodes, it follows that $p = O(P/\log P)$. Still, the cost of the MPP is not $\Theta(p)$ but $\Theta(P)$ as at least the state for each PRAM processor must be kept, and also many interconnection networks, such as the butterfly network used in the Fluent Machine and the SB-PRAM, have cost $O(l \cdot p) = O(P)$ [3]. While those considerations have been largely ignored for cases where cost of a processor is much larger than the memory or register cost to store PRAM processor states, it becomes relevant when processing cost is small as in the case of the GCA. A major disadvantage of this emulation is that each emulation step

³ Each processor-memory pair of the MPP is the source and the origin of at most l packets.

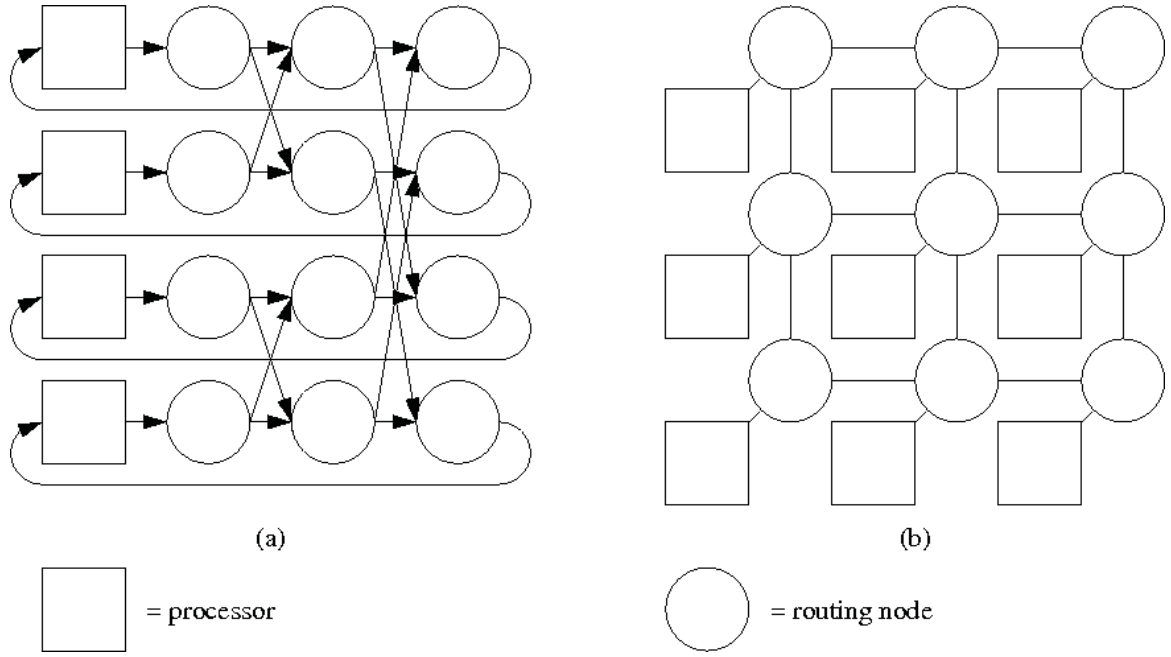


Fig. 3. PRAM emulations: (a) a coated butterfly network, (b) a processor mesh.

takes time $O(l)$, besides the cost being proportional to the number of PRAM processors simulated.

A different approach by Bilardi and Preparata [16] proposes to implement a PRAM on a P -processor network with latency l , e.g. a square mesh, so that each node has routing, processing, and storage functionality. An example is depicted in Fig. 3(b). The cost of the MPP is $O(P)$ as in the previous emulation strategy, and it can host P PRAM processors. While an emulation of shared memory access will take time $O(l)$ as before, simulation of PRAM steps without shared memory access can be done in time $O(1)$, as each MPP processor now only has to host one PRAM processor. The scheme above is formally shown to be optimal in [16]. Intuitively, one can argue that in the general case, all schemes asymptotically have the same cost and the same time span to implement one PRAM step, given that identical networks are used. Yet, in steps that only work locally or in a regular manner (e.g. each PRAM processor may access a memory location hosted by its right neighbor), the PRAM emulations above cannot use this fact to their advantage, even if they are aware of it. Yet, the Bilardi/Preparata scheme only needs a short time for this step.

The Bilardi/Preparata scheme also in a natural manner translates to GCAs, where each cell typically hosts computational and storage functionality, and where implementations complete each GCA step as fast as possible. Furthermore, many algorithms are regular enough so that a compiler can identify steps where concurrent and arbitrary access to the shared memory is not needed; all of these steps can be simulated fast. As thus, GCAs seem an optimal implementation platform for PRAMs.

5 Conclusions

We have presented how Global Cellular Automata provide a bridge between the theory of parallel algorithms, formulated in the PRAM model, and practical realizations in reconfigurable, massively parallel hardware. First, we argued that PRAM algorithms can be efficiently transformed into GCA algorithms. Second, we argued that optimality criteria for GCAs are both closer to reality, and more advantageous than in the PRAM model, thus allowing to use simpler algorithms. Third, we argued that typical implementations of GCAs in reconfigurable hardware are advantageous over classical PRAM emulations with their massive latency hiding, if the full power of concurrent access is not needed too often, which holds for many PRAM algorithms. Thus, we conclude that GCA implementations in reconfigurable hardware provide PRAM emulations tailored to the specifics of the algorithm to be executed.

References

1. JáJá, J.: An Introduction to Parallel Algorithms. Addison Wesley, Reading, MA (1992)
2. Karp, R.M., Ramachandran, V.L.: A survey of parallel algorithms for shared-memory machines. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, Vol. A. Elsevier (1990) 869–941
3. Keller, J., Kessler, C.W., Träff, J.L.: Practical PRAM Programming. Wiley & Sons (2001)
4. Valiant, L.G.: General purpose parallel architectures. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, Vol. A. Elsevier (1990) 943–971
5. Hoffmann, R., Völkmann, K.P., Waldschmidt, S., Heenes, W.: GCA: Global Cellular Automata. A Flexible Parallel Model. In: PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies, London, UK, Springer-Verlag (2001) 66–73
6. Jendrszok, 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) 1299–1316
7. Jendrszok, J., Hoffmann, R., Lenck, T.: Generated horizontal and vertical data parallel GCA machines for the N-body force calculation. In: Proceedings of the 22nd Conference on the Architecture of Computing Systems (ARCS). (2009) 96–107
8. Jendrszok, J., Lenck, T., Hoffmann, R., Osterloh, A., Keller, J.: A data parallel GCA machine adapted to the N-body force calculation. submitted (2009)
9. Heenes, W.: Entwurf und Realisierung von massivparallelen Architekturen für globale Architekturen. PhD thesis, Techn. Univ. Darmstadt, Dept. of Computer Science (2006)
10. Wyllie, J.C.: The Complexity of Parallel Computation. PhD thesis, Department of Computer Science, Cornell University (1979)
11. Cole, R., Vishkin, U.: Faster optimal parallel prefix sums and list ranking. Information and Computation **81**(3) (1989) 334–352
12. Leppänen, V., Penttonen, M.: Work-optimal simulation of PRAM models on meshes. Nordic Journal of Computing **2**(1) (1995) 51–69

13. Leighton, T., Maggs, B., Rao, S.: Universal packet routing algorithms. In: Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science. (1988) 256–269
14. Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M.: The NYU ultracomputer — designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers* **C-32**(2) (1983) 175–189
15. Ranade, A.G.: How to emulate shared memory. *Journal of Computer and System Sciences* **42**(3) (1991) 307–326
16. Bilardi, G., Preparata, F.P.: Horizons of parallel computation. *Journal of Parallel and Distributed Computing* **27**(2) (1995) 172–182

Network Optimization of a Multiprocessor Architecture for the Massively Parallel Model GCA

Christian Schäck, Wolfgang Heenes and Rolf Hoffmann

Technische Universität Darmstadt, FB Informatik, FG Rechnerarchitektur,
Hochschulstraße 10, D-64289 Darmstadt, Germany
{schaeck, heenes, hoffmann}@ra.informatik.tu-darmstadt.de
<http://www.ra.informatik.tu-darmstadt.de>

Abstract. The GCA (Global Cellular Automata) model consists of a set of cells which change their states synchronously depending on the states of their neighbors like in the classical CA model. In differentiation to the CA model the neighbors are not fixed and local, they are variable and global. The GCA model is applicable to a wide range of parallel algorithms. In this paper a general purpose multiprocessor architecture for the massively parallel GCA model is presented. In contrast to a special purpose implementation of a GCA algorithm the multiprocessor system allows the implementation in a flexible way through programming. The architecture mainly consists of a set of processors (Nios II) and a network. The Nios II features a general-purpose RISC CPU architecture designed to address a wide range of applications. The network used is a modified omega network supplemented with a register stage. Only read-accesses through the network are necessary in the GCA model leading to a simplified structure. A system with up to 32 processors was implemented as a prototype on an FPGA. The modified omega network avoids the decreasing clock frequencies as an effect of an increasing number of processors and an increasing network size. As an example bitonic sort was implemented on the multi-softcore system. A speed-up of 10 was reached for 32 cores. Bitonic sort was also compared to quicksort on a single processor.

Key words: Global Cellular Automata, multicore architecture, omega network, network on chip, FPGA, Nios II softcore

1 Introduction

The GCA (Global Cellular Automata) model [1] is an extension of the classical CA (Cellular Automata) model [2]. In the CA model the cells are arranged in a fixed grid with fixed connections to their local neighbors. Each cell computes its next state by the application of a local rule depending on its own state and the states of its neighbors. The data accesses to the neighbors states are read-only and therefore no write conflicts can occur. The rule can be applied to all cells in parallel and therefore the model is inherently massively parallel. The CA model is suited to all kind of applications with local communication, like physical fields, lattice-gas models, models of growth, moving particles, fluid flow, routing problems, picture processing, genetic algorithms, and cellular neural networks.

The GCA model is a generalization of the CA model which is also massively parallel. It is not restricted to the local communication because any cell can be a neighbor. Furthermore the links to the neighbors are not fixed; they can be changed by the local rule from generation to generation. Thereby the range of parallel applications is much wider for the GCA model. Typical applications besides the CA applications are graph algorithms [3], hypercube algorithms, logic simulation [4], numerical algorithms, communication networks, neuronal networks, and graphics.

The state of a GCA cell consists of a data part and one or more pointers (Fig. 1). The pointers are used to dynamically establish links to global neighbors. We call the GCA model one handed if only one neighbor can be addressed, two handed if two neighbors can be addressed and so on. In our investigations about GCA algorithms we found out that most of them can be described with only one link.

The aim of our research is the hardware and software support of this model. There are mainly three possibilities for an implementation.

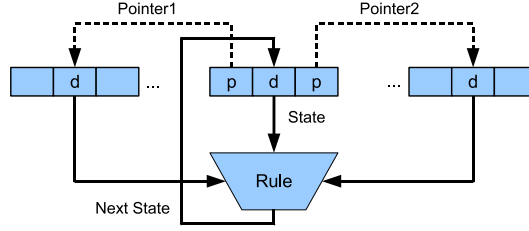


Fig. 1. The operation principle of the GCA

1. **Fully Parallel Architecture.** A specific GCA algorithm is directly mapped into the hardware using registers, operators and hardwired links which may also be switched if necessary. The advantage of such an implementation is a very high performance [5], but the problem size is limited by the hardware resources and the flexibility to apply different rules is low.
2. **Partially Parallel Architecture with Memory Banks.** This architecture [6–8] offers also a high performance, is scalable and can cope with a large number of cells. The flexibility to cope with different rules is restricted.
3. **Multiprocessor Architecture.** This architecture is not as powerful as the above two ones, but it has the advantage that it can be tailored to any GCA problem by programming. It also allows integrating standard or other computational models.

In [9] we described a 16-bit multiprocessor architecture with a multiplexer network, used for dedicated algorithms like fast fourier transformation and bitonic merge. In this contribution we are presenting a more powerful 32-bit multiprocessor architecture for the GCA model with an omega network [10]. It was implemented on two different Altera FPGAs using the Nios II softcore.

2 Multiprocessor Architecture

2.1 Overview

- The system consists of p processors with internal memories (program and data) and an omega interconnection network (Fig. 2).
- The data memories are implemented as dual port memories. One port is directly connected to its associated processor. The second port is connected to the network enabling all other processors to access the data in the memory.
- Each data memory holds a part of the GCA cell field of the application.
- A processor can modify only its own cells stored in its internal data memory.
- Each processor has only read access to the other data memories, write accesses need not to be implemented due to the GCA model.
- The local GCA rule is programmable by processor instructions.
- The processor instructions support read and write accesses to the cells in the internal memory and the read accesses to external cells stored in the other memories.

One processor with its associated dualport data memory is called *Processing Unit* (PU). The omega network interconnects the processing units. Therefore the network complexity depends on the complexity of the communication patterns which is needed for the class of GCA algorithms to be implemented. For many GCA algorithms [11] the communication pattern is rather simple which simplifies the design of the network. Simple or specialized networks can be implemented with multiplexers or fixed connections [9, 10], complex networks have to be able to manage concurrent read accesses to arbitrary external memory

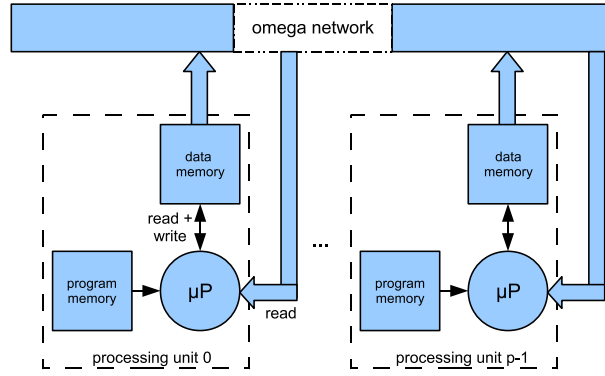


Fig. 2. General system architecture with datapath

locations. As in the GCA model a cell is not allowed to modify the contents of another cell, the network design is simplified because write accesses need not to be implemented. Memory arbitration of concurrent read accesses is handled decentralized within the switching elements of the omega network. Thereby concurrent read accesses need not to be handled by each PU which reduces the overall control logic. Therefore it makes no difference if a PU has to wait due to a blocked network or due to a concurrent read access.

2.2 The Nios II Softcore Processor

The Nios II processor [12] is a general-purpose RISC processor core, providing:

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- Dedicated instructions for computing 64-bit and 128-bit products of multiplication
- Floating-point instructions for single-precision floating-point operations
- Custom instructions

For the implementation of the multiprocessor architecture, the custom instructions [13] turned out to be very useful (see section 2.4).

2.3 The Omega Network

The omega network is commonly used in parallel computer systems. It belongs into the category of blocking networks. We chose the omega network as it can be used without blockings for accesses following a pattern to the power of two. Figure 3 shows the possible switching states of one switching element. The switching elements are able to process a broadcast request. If two processing units want to access the very same memory location, the concurrent access can be resolved with broadcasting.

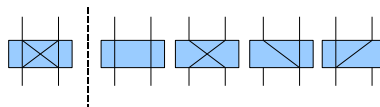


Fig. 3. Switching element, four possible switching states

Figure 4 shows the control and data signals of a switching element. Input 0 (ain_0 , req_0) has higher priority compared to input 1 (ain_1 , req_1). The switching elements consist of combinatorial logic only.

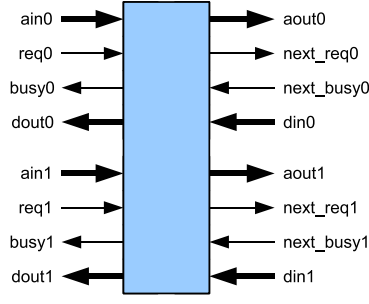


Fig. 4. Control and data signals of a switching element: ain/aout = address in/out, din/dout = data in/out, req/next_req = read request send to the next switch, busy/next_busy = acknowledge send to the previous switch

A big issue is the performance of the network. With an increasing number of processors the clock frequency degrades. This is due to the large combinatorial network with multiple critical paths. All paths in the network are considered as being critical as they all contain the same amount of switching elements and therefore all paths have to be treated in the same way to achieve a higher clock frequency. In order to reduce the length of the critical paths, registers can be inserted into the network. Thereby the performance degrades for all accesses via the network, because four clock cycles are necessary for one access, but the overall system clock frequency is increased. We have investigated the omega network with a register stage directly in the middle of the network. If there is an odd amount of switching elements in the network the register stage is placed closer to the data memories.

Figure 5 shows the multiprocessor architecture with eight processing units. For better understanding of the architecture the processing units are shown twice. Concurrent accesses to the same memory location are sequentialized through the two priorities of the inputs of the switching elements.

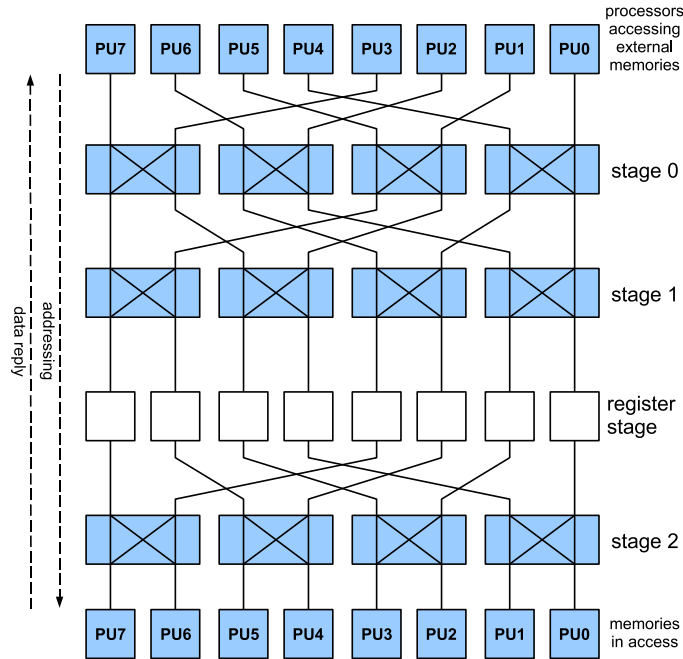


Fig. 5. Multiprocessor architecture with eight processing units and register stage

2.4 Custom Instructions

We expanded the Nios II instruction set with custom instructions and additional hardware. The custom instructions are:

- INTERNAL_GET_P: read the pointer p (Fig. 1) from internal memory
- INTERNAL_GET_D: read the data d from internal memory
- SET_PS: shift pointer p right by one and write to the internal memory
- SET_P: write pointer p to the internal memory
- GET_P: read pointer p by using the network
- SET_D: write data d to the internal memory
- GET_D: read data d by using the network
- NEXTGEN: processor synchronization command (see Section 2.5)

2.5 Blocking and Synchronization

The omega network is a blocking network. Using eight processors (Fig. 5) it is possible that up to seven of them will be blocked during a concurrent read access to eight different memories. This leads to unsynchronized processor execution of the cellular field. As a consequence of the partially sequential calculation two memories (old/new generation) are needed and data consistency has to be guaranteed. Due to the blocking read accesses the processors might calculate different generations at a time. To avoid such a behavior the processors need to be synchronized at generation transition which is done by using a special synchronization command at this state. A barrier-synchronization technique [14, p. 165] is implemented by *AND* gating. Generation synchronization cannot be skipped for consecutive generations without external read accesses as the generation synchronization also switches between the two memories.

3 FPGA Prototype Implementation

3.1 FPGA Implementation - Synthesis results

We will now compare the synthesis results for the architecture with a register stage in the network against an architecture without a register stage.

The first prototyping platform is a Cyclone II FPGA with the Quartus II 8.1 synthesis software from Altera. The Cyclone II FPGA contains 68,416 logic elements (LE) and 1,152,000 RAM bits [15]. The implementation language is Verilog HDL. The Nios II processor is built with the SOPC-Builder. We used the Nios II/f core that has the highest execution performance and uses a 6-stage pipeline. The DMIPS¹ [16] rate is 218 at the maximum clock frequency of 185 MHz. The synthesis settings are optimized for speed. Table 1 shows the numbers of logic elements (LE), logic elements for network, the memory usage, the maximum clock frequency and the estimated max. DMIPS rate for all processors [12, Chapter 5]. With the DMIPS rate the DMIPS speed-up can be calculated. The DMIPS speedup does not take into account the possible slowdown induced by communication between the processors. It is used to estimate the performance that can be achieved with the different amount of processors. To compare the architectures (table 1, table 2) the columns A through E have been calculated according to the following equations: $A = B + C + \delta$ (δ = additional logic²), $D = C/p$, $E = C/B$. Column B shows the resource usage for the processor subsystem.

¹ Dhrystone Million instructions per second

² 32 bit timer, custom instruction decoding, memory controller, processor connections, generation and read synchronization

	A	B	C	D	E	F	G	H	I	J
p	total LE	LEs for processors	LEs for network	network per processor	network over processor LEs	memory bits	register bits	max. clock (MHz)	max. DMIPS	DMIPS speed up
1	2,824	2,824	-	-	-	195,296	1,585	145	168	1.0
4	10,165	8,734	154	38.50	1.5%	289,568	4,910	105	487	2.8
8	19,985	16,930	866	108.25	4.5%	415,264	9,351	75	696	4.1
16	40,878	33,943	2,145	134.06	5.5%	666,656	18,244	55	1,020	6.0

Table 1. Resources and clock frequency for the Cyclone II FPGA **without** register stage

	A	B	C	D	E	F	G	H	I	J	K
p	total LE	LEs for processors	LEs for network	network per processor	network over processor LEs	registers for network	memory bits	register bits	max. clock (MHz)	max. DMIPS	DMIPS speed up
1	2,824	2,824	-	-	-	-	195,296	1,585	145	168	1.0
4	10,029	8,734	416	104	4.8%	181	289,568	5,091	105	487	2.8
8	19,699	16,930	1,113	139.125	6.6%	361	415,264	9,712	90	835	4.9
16	40,445	33,943	3,276	204.75	9.7%	721	666,656	18,965	70	1,299	7.7

Table 2. Resources and clock frequency for the Cyclone II FPGA **with** register stage

Comparing the two tables (table 1, table 2) shows a performance improvement for the 16 and 8 processor implementation with a registered network against the implementation without register stage. This is explained by the increased clock frequency. For 1 and 4 processors there is no difference in the DMIPS rate because the clock frequency is the same. The additional clock cycles needed for a network access are not reflected in the DMIPS rate. The DMIPS speed-up for 16 processors is about 28% higher with the register stage.

The second prototyping platform was a Stratix II FPGA with 143,520 ALUTs, 71,760 ALMs and 9,383,040 RAM bits [17]. Table 3 shows the required resources. With 32 processors about 30 percent of the FPGA's ALUT resources are used. The synthesis settings were balanced between area and speed. Synthesis for 64 processors generally seems to be achievable but could not be accomplished due to problems with the synthesis software.

	-	A	B	C	D	E	F	G	H	I	J
p	total ALMs	total ALUTs	ALUTs for processors	ALUTs for network	network per processor	network over processor ALUTs	memory bits	register bits	max. clock (MHz)	max. DMIPS	DMIPS speed up
1	1,203	1,592	1,592	-	-	-	195,296	1,520	200	232	1.0
4	4,147	5,354	4,683	279	69.75	5.9%	289,568	4,813	133	617	2.6
8	8,397	10,634	9,006	818	102.25	9.0%	415,264	9,200	100	928	4.0
16	17,448	21,439	17,697	2,344	146.50	13.2%	666,656	17,994	75	1392	6.0
32	36,131	43,815	34,404	6,092	190.38	17.7%	1,169,440	35,591	55	2041	8.7

Table 3. Resources and clock frequency for the Stratix II FPGA **without** register stage

As for the Cyclone platform there is no difference between the unregistered and registered implementations for 1 and 4 processors on the Stratix II platform. The registered network has the highest benefit for the 32 processor implementation as it has the largest network. The 32 processor implementation with register stage has a 37% higher DMIPS rate than the implementation without the register stage.

	-	A	B	C	D	E	F	G	H	I	J	K
p	total ALMs	total ALUTs	ALUTs for processors	ALUTs for network	network per processor	network over processor ALUTs	registers for network	memory bits	register bits	max. clock (MHz)	max. DMIPS	DMIPS speed up
1	1,203	1,592	1,592	-	-	-	-	195,296	1,520	200	232	1.0
4	4,355	5,437	4,683	326	81.50	6.9%	184	289,568	4,995	133	617	2.6
8	8,344	10,780	9,006	922	115.25	10.2%	368	415,264	9,568	120	1113	4.8
16	17,638	21,935	17,697	2,616	163.50	14.8%	736	666,656	18,726	95	1763	7.6
32	36,577	44,441	34,404	6,637	207.40	19.3%	1,472	1,169,440	37,047	75	2784	12.0

Table 4. Resources and clock frequency for the Stratix II FPGA **with** register stage

As shown in the tables (column E) the network ALUTs are relatively low compared to the processor ALUTs (19% for 32 processors, STRATIX II). Therefore the architecture can be scaled up to more than 128 processors (by extrapolation) using the same amount of logic for the processors as for the network.

Comparing the overall LE/ALM/(ALUT) usage shows that the register stage implementation does not need mentionable more resources. In fact it needs less resources on the Cyclone II while it needs more resources on the Stratix II. The LEs of the Cyclone II can be used more efficiently than the ALMs/ALUTs of the Stratix II. For example the Cyclone II supports register packing [15], allowing to use the LUT and the register of a LE for unrelated functions.

4 Application Bitonic Sort

The principle of the multiprocessor architecture will be demonstrated by the bitonic sort algorithm. In the GCA model each cell holds its own value and compares it with a neighbor's cell which is in changing distance. The distance is a power of two. To sort an unsorted field of numbers the bitonic merge algorithm (listing 1.1, line 13-29) is executed in a loop (line 9). Within each new loop execution the maximum distance to the neighbors increases by the power of two (line 7). If for example the bitonic merge algorithm was executed for the distances 1, 2 and 4 it will be executed on the distances 1, 2, 4 and 8 in the next round. But not only the distances increases. The direction (ascending or descending) changes accordingly to the distance (line 23, 26). The sequential time complexity of the complete sorting algorithm is $O(N \log^2 N)$. For a parallel implementation on $p = N$ processors the parallel time complexity of the algorithm is $O(\log^2 N)$

Listing 1.1. C program for bitonic sort with custom instructions

```

0 void bitonicSort()
1 {
2     int j, ii, N, I, k, neighbour, js;
3
4     for(j=1; j<DAT_BLK; j=j<<1)
5     {
6         k=j;
7         js=j<<1;
8
9         while(k>=1)
10        {
11            for(ii=SC; ii<LOCL_BLK+SC; ii++)
12            {
13                if(j!=k)
14                    k=ALT_CI_CUSTOM_NETWORK_EXT_INST(INTERNAL_GET_P, ii, 0);
15
16                if((ii & k)==0) neighbour=ii+k;
17                else neighbour=ii-k;
18
19                N = ALT_CI_CUSTOM_NETWORK_EXT_INST(GET_D, neighbour, 0);
20                I = ALT_CI_CUSTOM_NETWORK_EXT_INST(INTERNAL_GET_D, ii, 0);
21
22                if((ii & k)==0)
23                    if(!(ii&js)==(N<I)) ALT_CI_CUSTOM_NETWORK_EXT_INST(SET_D, ii, N);
24                    else ALT_CI_CUSTOM_NETWORK_EXT_INST(SET_D, ii, I);
25                else
26                    if(!(ii&js)==(I<N)) ALT_CI_CUSTOM_NETWORK_EXT_INST(SET_D, ii, N);
27                    else ALT_CI_CUSTOM_NETWORK_EXT_INST(SET_D, ii, I);
28
29                ALT_CI_CUSTOM_NETWORK_EXT_INST(SET_PS, ii, k);
30            }
31
32            ALT_CI_CUSTOM_NETWORK_EXT_INST(NEXTGEN, 0, 0);
33            k=ALT_CI_CUSTOM_NETWORK_EXT_INST(INTERNAL_GET_P, SC, 0);
34        }
35    }
36 }
37
38 int main()
39 {
40     ALT_CI_CUSTOM_NETWORK_EXT_INST(NEXTGEN, 0, 0);
41     ALT_CI_CUSTOM_NETWORK_EXT_INST(NEXTGEN, 0, 0);
42     bitonicSort();
43     return 0;
44 }
45

```

The system was implemented for $p = 1, 4, 8, 16, 32$ processors and $N = 2048$ cells. Each processor holds $n = N/p$ cells. The clock cycles have been measured using a dedicated hardware timing register triggered via the custom instruction.

The execution time is the number of clock cycles over the clock frequency. Table 5 shows a linear speed-up for the clock cycles needed to execute the program for $N = 2048$. There are no blocking accesses to the network caused by the bitonic sort algorithm and therefore the cycles needed show a linear speed-up.

Considering the degradation of the clock frequency for 32 processors, a real speed-up³ of about 10.2 is reached. Comparing these values to table 3 and table 4 shows that the real speed-up is in between the two estimated speed-ups calculated out of the DMIPS rate.

processors	cycles	cycle speed-up	execution time	real speed-up
1	4,971,726	1.00	24.8 ms	1.0
4	1,443,694	3.4	10.8 ms	2.3
8	726,336	6.8	6.0 ms	4.1
16	366,650	13.5	3.8 ms	6.4
32	182,598	27.2	2.4 ms	10.2

Table 5. Clock cycles, execution time and speed-up (Stratix II)

Compared with two quicksort implementations [18, p. 145 et seqq.] [19, p. 153 et seqq.] with a better complexity of only $O(N \log N)$ implemented in C on one Nios II/f processor we get the results shown in table 6. The median quicksort algorithm chooses, in contrast to the standard implementation, the pivot element out of three elements.

³ Real Speed-up: Speed-up for an application on a real machine

algorithm	sorted data		random data	
	cycles	execution time	cycles	execution time
standard quicksort	23,314,342	169.56 ms	502,411	3.65 ms
median quicksort	420,197	3.06 ms	542,098	3.94 ms

Table 6. Clock cycles and execution time for two quicksort implementations on one processor

The quicksort algorithm has a data dependent performance. In the case of sorted data the standard quicksort implementation is much slower than the bitonic sort algorithm whereas it is generally faster in the best case scenario (keeping in mind the different complexities of the algorithms and the amount of processors). In the case of random data the quicksort implementations are about as fast as the bitonic sort using 16 processors. The advantage of using the bitonic sort algorithm is that it is data independent and therefore it is running within a fixed time for a given size of values. It also maps better onto the GCA architecture using it very efficiently.

The resulting performance of the bitonic sort GCA-algorithm on the multiprocessor architecture is not really convincing compared to the quicksort algorithm on one processor. The main reason is the high work-complexity of $O(N \log^2 N)$ of the bitonic sort algorithm. As the number p of processors is smaller than N the optimal parallel time complexity of $O(\log^2 N)$ for $p = N$ could not be reached. A better execution time might be reached if the processors first use an efficient serial sort algorithm on sublists which are later merged between the processors.

5 Conclusion

A programmable multiprocessor architecture for the massively parallel GCA model was designed and implemented as a prototype in FPGA technology. The architecture consists of p processing units with internal memories and a read-only interconnection network.

Compared to a dedicated implementation the proposed architecture is very flexible because it can be easily adapted to different GCA algorithms by programming. The speed-up of the prototype increases with the number of processors for the investigated algorithm. A DMIPS speed-up of 12 could be reached by increasing the number of processors from 1 to 32. The speed-up regarding clock cycles is 27.2 while the real speed-up is 10.2 comparing the 1- against the 32-processor implementation. The insertion of the registers in the network increased the real speed-up for bitonic merge from 8.2 to 10.9 for the 32 processor version on the Stratix II [20]. This shows that although each network access uses an additional clock cycle the performance increases because the increased clock frequency has a higher impact on the system performance. For a large number of processors the registered omega network should be used. Using other networks, e.g., without read synchronization and further optimizations techniques such as pipelining to keep the overall system frequency high is a subject under investigation. We are also using the multi-softcore system to accelerate the simulation of multi-agent systems.

Acknowledgement. We would like to thank Altera for their technical support on software issues that occurred during the implementation.

References

1. Hoffmann, R., Völkman, K.P., Waldschmidt, S., Heenes, W.: GCA: Global Cellular Automata, A Flexible Parallel Model. In: 6th International Conference on Parallel Computing Technologies (PaCT). (2001)
2. von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, Urbana and London (1966)
3. Johannes Jendrszczok and Rolf Hoffmann and Jörg Keller: Implementing Hirschbergs PRAM-Algorithm for Connected Components on a Global Cellular Automaton. In: 21nd International Parallel & Distributed Processing Symposium (IPDPS), 9th Workshop on Advances in Parallel and Distributed Computational Models. (2007)
4. Wiegand, C., Siemers, C., Richter, H.: Definition of a configurable architecture for implementation of global cellular automaton. In: 17th International Conference on Architecture of Computing Systems (ARCS 2004). (2004)
5. Heenes, W., Hoffmann, R., Kanthak, S.: FPGA Implementations of the Massively Parallel GCA Model. In: International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing (WMPP). (2005)
6. Heenes, W., Völkman, K.P., Hoffmann, R.: Architekturen für den globalen Zellularautomat. In: 19. PARS Workshop, Gesellschaft für Informatik (GI). (2003)
7. Hoffmann, R., Völkman, K.P., Heenes, W.: GCA: A massively parallel Model. In: International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing (WMPP). (2003)
8. Jendrszczok, J., Ediger, P., Hoffmann, R.: A Scalable Configurable Architecture for the Massively Parallel GCA Model. In: 22nd International Parallel & Distributed Processing Symposium (IPDPS), 10th Workshop on Advances in Parallel and Distributed Computational Models (APDCM). (2008)
9. Heenes, W., Hoffmann, R., Jendrszczok, J.: A Multiprocessor Architecture for the Massively Parallel Model GCA. In: International Parallel & Distributed Processing Symposium (IPDPS), Workshop on System Management Tools for Large-Scale Parallel Systems (SMTPS). (2006)
10. Heenes, W.: Entwurf und Realisierung von massivparallelen Architekturen für Globale Zellulare Automaten. PhD thesis, Technische Universität Darmstadt (2007)
11. Heenes, W.: Globaler Zellularer Automat: Algorithmen und Architekturen. Master's thesis, Technische Universität Darmstadt (2001)
12. Altera, NIOS II Website.
<http://www.altera.com/products/ip/processors/nios2/ni2-index.html> (2009)
13. Altera, NIOS II Website.
<http://www.altera.com/literature/lit-nio2.jsp> (2009)
14. Ungerer, T.: Parallelrechner und parallele Programmierung. Spektrum Akademischer Verlag, Heidelberg, Berlin (1997)
15. Altera, Datasheet Cyclone II.
http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf (2006)
16. Weiss, A.R.: Dhrystone benchmark - history, analysis, "scores" and recommendations. <http://www.ebenchmarks.com/download/ECLDhrystoneWhitePaper.pdf> (2002)
17. Altera, Datasheet Stratix II.
http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf (2006)
18. Sedgewick, R.: Algorithmen in C. Addison-Wesley, München, Boston, San Francisco (1992)
19. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. 24. edn. The MIT Press, Cambridge, Massachusetts, London, England (1990)
20. Schäck, C., Heenes, W., Hoffmann, R.: A Multiprocessor Architecture with an Omega Network for the Massively Parallel Model GCA. In: International Symposium on Systems, Architectures, Modeling and Simulation (SAMOS IX Workshop). (2009)

Architecture and Implementation of an MMU on a Server-class Infiniband HCA

Andreas C. Döring

ado @ zurich.ibm.com

IBM Research GmbH, Zurich Research Laboratory

Säumerstrasse 4

CH-8803 Rüschlikon, Switzerland

Abstract. IBM's System z® and System p® servers use Infiniband® for IO, clustering, and coupling. The recent families z10™ and POWER6™ use several chips in common, including a second-generation two-port Double Data Rate (DDR) 12x Infiniband Host Channel Adapter (HCA). This chip is manufactured in 90-nm technology and implements the InfiniBand standard and a proprietary protocol fully in hardware. The required address translation is implemented in an on-chip Memory Management Unit (MMU). The architecture, logic design, and verification of this unit are described in this paper.

1 Introduction

The Infiniband standard [1] is the result of the combined effort of two initiatives, Futurebus and Next Generation IO, which both inherit the concept of direct communication of an application with IO hardware in the Virtual Interface Architecture. Infiniband-based products are successful as cluster interconnects and for interconnecting servers as well as with storage products in the data center. The high performance is achieved by the use of data exchange through shared memory regions and triggering actions in the hardware by writing to so-called doorbell registers by the applications directly. Sharing memory between a virtual-addressing application and CPU-independent hardware components requires the implementation of address protection and translation mechanisms similar to those found in the processor. A unit implementing this is called a Memory Management Unit (MMU). For the high data rate required by clustering and IO in multiprocessor servers, a high rate of address checks and translations needs to be provided. High-end servers require logic partitioning. This impacts the MMU architecture, because memory needs to be managed on three levels, namely the application, the guest operating system and the hypervisor. Furthermore, the reliability expectations for servers require strong protection against hardware and software errors. To reduce design cost, the described Infiniband HCA is used in two IBM server families, System z (Mainframes) and System p (POWER-6 based UNIX ® servers) [2]. Both families have common requirements with respect to performance and quality, but also specific requirements for each individual family. This paper describes the micro-architecture

and logic design of such an MMU. The chip is implemented in IBM's 90-nm copper technology, mainly using a standard cell library design style.

The microarchitecture of MMUs in the IO domain has not been published so far, even though the knowledge of an MMU's architecture, and operation, in particular related to caching has a considerable impact on system performance. Performance studies using Infiniband so far only could speculate about these details.

It is assumed that the reader is familiar with Infiniband, as we can give here only a very short introduction.

Infiniband uses queues to communicate between the network hardware and the application. After setup, the operating system is not needed for sending or receiving data. Data is sent by placing a so called Work Queue Element (WQE) into one of many send queues. The send queues are organized as contiguous memory region in the virtual address space, where for reading and writing the address is wrapped at the end to the beginning. To trigger sending, the application writes the number of new WQEs in the send queue into a register of the hardware device, the so called doorbell. By address mapping in the host processor each application has a separate doorbell register. The fifos may not be contiguous in real memory, but by using data types such as linked lists, the MMU presented here is not needed for queue access. The WQEs contain information about the packet, such as operation, destination node and queue, and one or several descriptors of data segments that contain the payload of the packet. Each data descriptor contains a virtual start address, length and a key. This memory key is obtained once by registering a memory region with the Infiniband device for use for sending or receiving. The registration "pins" the memory (inhibits swapping it out) and builds the address translation data structures described below. Receiving works similarly, a WQE can be used to determine where received data should be stored. For RDMA (Remote Direct Memory Access) operations the data descriptors are provided by the send side and do not come from a receive WQE. In summary, the more and smaller the data segments are the more address translation have to be carried out. The system design target of the described Infiniband HCA chip was to make maximum use of the available memory access bandwidth. Using 4K data segments this results in 1.75 million address translations for send and for receive traffic.

At the time when the high-level design was done no information on expected workloads was available. Even now, we have only limited insight how the requirements of important applications are for the MMU.

For the descriptions of data structures and algorithms a C-like pseudo-code is used.

In Section 2 the data structures and architected operation of the MMU are outlined. Section 3 describes the detailed design and verification approach. In the final section, lessons learned from the project are summarized.

2 Address Protection and Translation

Infiniband in a partitioned server requires three levels of protection management, namely, the hypervisor, the guest operating system, and the application. This is reflected in the data structure and operation of the MMU illustrated in Fig. 1: On the hypervisor level, an on-chip memory contains a set of registers. These registers describe a set of pages by providing the page address, error state, and owning partition. Each guest OS manages the keys contained in the pages they receive from the hypervisor. Each page contains the information for 64 keys. The keys can be used as R-keys or L-keys (Remote/Local), for memory regions or windows.

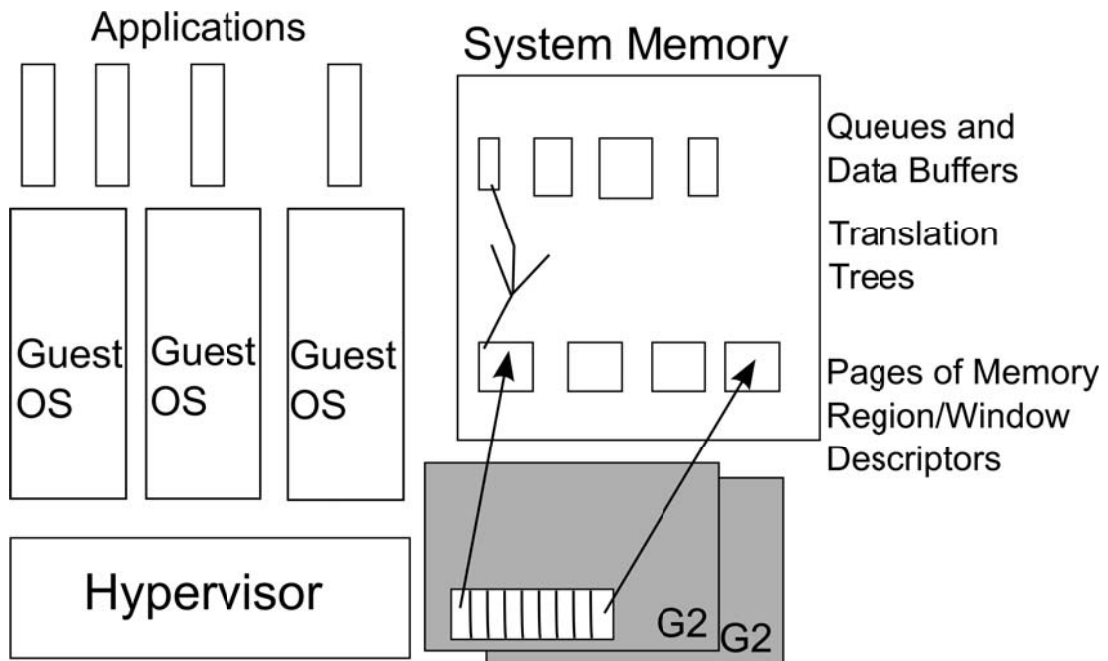


Fig. 1. System hierarchy and data structures for address translation, the data structures correspond to the management levels

The following registers are implemented inside the MMU (uint n indicates an n -bit unsigned integer):

```
const int pagenum=2048;
uint8[pagenum] LPAR_IDs;
uint8 CTRL;
uint3 keyctrl[64];
bool[pagenum] LPAR_valid;
uint8[pagenum] SW_ERROR;
uint64[pagenum] HW_ERROR;
keydescriptor*[pagenum] page_ptr;
```

The descriptor of a memory region or window is as follows:

```
struct {
    uint64 base_addr, length;      // in Bytes
    uint64 region_base;           // only used for windows
    uint32 protection_domain;      // defined by Infiniband
    uint8 key_instance;
    uint8 format; // number of tree levels, page size
    bool[5] permissions;          // e.g. write, bind
    bool valid; bool window;
    bool LorR;
    char *tree_root[4];
    uint region_key; // needed only for windows
} keydescriptor;
```

A descriptor consumes 64 Bytes of main memory, thus aligning well with the cache line granularity (128 Bytes for p- and 256 Bytes for z-Series). A set of 512 keys, called static keys, plays an important role for performance: As each static key has a dedicated cache entry, their **keydescriptor** needs to be read from memory only once after creation. To allow distribution of these keys to several guest OSs, they are distributed to the 64 first pages with 8 keys each. The remaining 48 keys in these pages are invalid. Static keys can be used for regions, not for windows.

A translation request can be received either from the send or the receive processing units. The operation of a translation is as follows:

```
translate(uint32 key,
    bool is_read,
    bool is_rkey,
    char* vaddr, // virtual address
    uint16 length,
    uint8 LPAR_ID, // which guest partition
    uint32 protection_domain)
return
    (char* page0, // page address on 4K granularity
    char* page1, // optional second address
    uint8 error_code) // 0 for successful translation
{
    bool static_key = key < 0x100000;
    uint ix=key/0x4000;
    if (key > MAX_KEY ||
        (static_key && (key & 0x3F00 >800)) ||
        (!LPAR_valid[ix]) || (LPAR_IDs[ix] != LPAR_ID) ||
        (CTRL[ix]&6 != 4)) // not enabled or error state
        return error;
    keydescriptor kd= *page_ptr[key/0x400];
    if (!kd.valid ||
```

```

    (static_key & kd.window) ||
    (vaddr < kd.base_addr) ||
    (vaddr+length < kd.base_addr + kd.length) ||
    (key & 0xFF != kd.key_instance) ||
    (protection_domain != kd.protection_domain) ||
    (is_rkey != kd.LorR) ||
    !match(kd.permissions, is_read))
    return process_error(key); // updates SW_ERROR, HW_ERROR, CTRL
levels=kd.format&7; // 0 to 3 levels
pagebits = 12+4*(kd.format/8); // log of page size
pagemask = !(1<<(pagebits-1));
pageoffset = (vaddr & pagemask) - (kd.base_addr & pagemask);
// including range check:
p=kd.tree_root[pageoffset/(1<<(pagebits+9*levels))];
for (int l=0; l<levels; l++)
// 512 pointers per tree node
    {p=*p[pageoffset/((1<<(pagebits+9*i)))];
      p1=*p[pageoffset/((1<<(pagebits+9*i))+1)];} // if not last
p1=next4K(vaddr, p, p1, pagebits);
return (p, p1, 0);
}

```

As can be seen, there are authentication tests on the hypervisor level (correct partition) as well as on the guest OS level (correct protection domain). Up to four memory accesses are needed for one translation: the first for reading the keydescriptor, and up to three to walk through the translation tree. Each access can cause further errors, such as uncorrectable errors in the memory. Details of error processing are omitted here.

The second operation of the MMU is **bind**, which fills a **keydescriptor** such that it represents a window within a region. Information from both the region's key descriptor and from the bind work request are combined. As bind writes to a page of key descriptors, consistency with subsequent translations has to be ensured.

To achieve a high performance, several caches inside the MMU reduce the number of memory reads. As mentioned above, the static keys are directly mapped to a cache area. The content is read during the first translation, and an entry becomes only invalid by disabling (write 0 to CTRL register). For the remaining key descriptors, a cache of 1K entries is provided. The cache is fully associative and uses a random cast-out method that protects the entries most recently used. Explicit flushing by writing to a control register is possible, which is needed for resizing a memory region or software binding of windows. Another cache associates virtual addresses with two successive physical addresses that each saves entire tree walks. For the static entries four virtual addresses are provided for each key descriptor, whereas for the non-static keydescriptors only

one virtual address is provided. The addresses are on page-size basis, and can therefore cover a large address range if large page sizes are used.

Finally, a small cache saves some of the inner nodes of the translation tree, based on the requesting unit. Therefore, the inner nodes of the tree can be reused from the cache when a large memory region is being transferred, for instance, by an RDMA transfer. The cache relies on the policy that a given queue is processed on the same receive or send engine, if possible.

Here is an example:

A send operation needs the translation of address 0x72510300 for 256 bytes with key 0x0141733, a protection domain value of 0x77460ac1 is also provided. This key indexes the page 0x50, within the page entry 0x17 (6 bits). The lower 8 bits 0x33 are the key instance and allow the reuse of the same key descriptor. When the MMU receives the request it first checks that the key is valid, i.e. that the page index (0x50) is lower than the highest implemented page number (0x7ff for 2048 pages), and that the key is not 0x0. As the next step the MMU checks whether the page with index 0x50 is enabled and not in error state and whether the entry within the page is enabled. This is done by reading on-chip registers CTRL and keyctrl. Since the key is not static, as a next operation the CAM of the protection cache is searched using the key without the instance field (0x01417). Assume that the key_descriptor is found in the cache because of a previous address translation with the same key. The given virtual address (0x72510300) is compared to the starting address (say 0x7250080), and the sum of the virtual address and the length (0x72510400) is compared with the end of the region. The latter is determined by adding the starting address of the region and the length of the region. Furthermore the protection value from the key descriptor is compared against the parameter of 0x77460ac1. As a next step the page index needs to be found. For this example we use a page size of 64K. Both, the starting address of the region and the virtual address are truncated to 64K alignment and the difference is calculated ($0x72510000 - 0x72500000 = 10000$). This means that the required data is in the second page of this region. If the given region's length is smaller than $64K * 4 - 0x80$, four pages are sufficient and can be stored directly in the key_descriptor. Given this case, the offset within the page (0x300) is added to the real address of the second page (`tree_root[1]`). This value is returned to the send processor, thereby finishing this translation operation.

3 Design and Verification

To achieve high performance, up to eight operations can be carried out in parallel, where an operation can be either a translation, a bind, or a cache entry flush. Independent of this, registers of the MMU can be read or written. Parallel operation requires additional measures for consistency. For simplicity, parallel translations have to operate in disjoint keys, where for the bind operation both the region and the window are considered as "used". Before a requested operation can be started, a check is therefore carried out whether the key(s) of the

requested operation are currently in use. The parallel operations are organized as slots that consist of a set of approximately 200 latches. This slot stores the requested operation and its parameters, intermediate values such as the current level, and error codes. A new operation can be started if a slot is free, no currently used slot has the current key in use, and the key is not currently cast out. Only one request is tested per cycle. If an operation is accepted, the slot values will be filled in. The individual steps of an operation are structured in sub-units, organized along major steps or resources, such as search in the CAM for a key, or processing an error, which includes determination of the values for the HW_ERROR and SW_ERROR registers. Between some subunits direct connections exist, but whenever different paths can lead to the same operation (e.g., reading of the key descriptor cache array either after CAM lookup or directly after key validity checks) an arbitration is done, see Fig. 2.

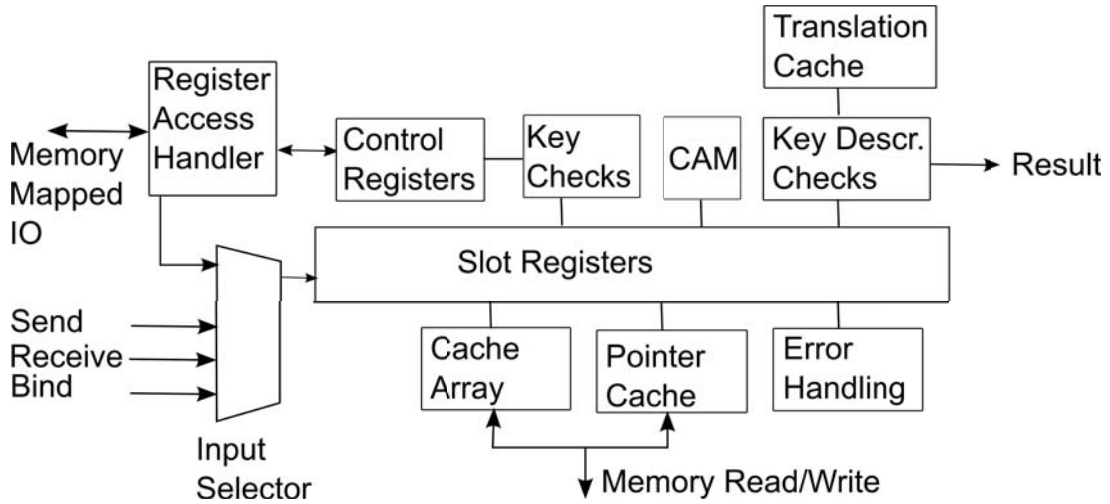


Fig. 2. Micro Architecture of the MMU

This approach has the advantage of offering easy scalability of the number of parallel operations, comparatively easy debugging during design and in the lab during hardware bringup, and good design partitioning to several designers. However, if the MMU is used mainly with the same type of operations, such as translation with non-static keys, several redundant arbitration cycles are inserted. If an external memory read is needed, the memory latency will dominate the translation latency, and extra arbitration cycles will be negligible. However, if most operations are operations with static keys, with either zero translation levels or with a hit in the translation cache, 3 of 13 cycles are arbitration. If higher emphasis is given to this best case, the design can be modified such that more sub-operations are done in parallel, reducing the translation latency to five cycles. The consequence would be an increased design complexity and a reduced throughput in case of translation errors, but the area would be nearly unchanged. Similar considerations apply to translations with non-static keys running from

caches, but the impact is smaller as the sequential dependency of the cache array read on the result of CAM lookup cannot be avoided.

To increase performance, the MMU can do a look-ahead processing after a translation. If a translation reaches the end of a page and uses at least one level of indirection, the MMU performs a translation of the next address to fill its caches. Therefore, if a later packet continues with the next virtual address soon, it will find the translation in the cache (either in the virtual-to-physical or the intermediate node cache).

The design is implemented in VHDL using IBM internal tools for synthesis, simulation, and physical design. Verity was used for ensuring synthesis correctness. Verification combined random-driven simulation and deterministic test cases, the latter mostly for error scenarios. The HCA portion of the chip runs at 250 MHz. Timing closure was reached in two steps: synthesis and a first timing check was done with a timing target of 3.2 ns to allow for clock skew and wiring, and a second timing analysis was done after placement and wiring. Only few paths in the MMU violated the first timing check. They related to a SRAM-read and Error-Correcting Code(ECC). As the ECC logic can be placed very close to the SRAMs, these violations were compensated in the wiring step.

The bringup was performed using bare-metal Linux® on p-Series and 370-architecture-based Linux on z-Series machines.

4 Conclusion

Several approaches to the implementation of a complex protocol, such as Infiniband, have been proposed in literature and used in products, ranging from software on a more or less specialized processor to a full hardware implementation as presented in this paper, e.g. [3]. Some well-known or obvious rules were once again confirmed by this design:

- Knowledge of the application and the architecture can greatly improve performance [4]; in this case the use of static keys for frequently used regions improves performance, even in cases where the cache capacity of the non-static keys is not exceeded and no cast-outs are needed.
- The use of large page sizes greatly increases the effectiveness of caches. Already the move from 4K to 64K pages provides a significant advantage. Note that between processor and IO the page sizes do not have to be the same.
- The hardware assumes that data is typically transferred in increasing order. SW can profit from it by also doing so.
- Address translation is faster if buffers do not cross page boundaries.
- To allow sufficient parallelism in the MMU, demanding applications should use a set of keys, e.g., three or four keys. Using too few keys causes stalls in the request acceptance step, whereas too many keys will reduce cache effectiveness.

Several further improvements were considered, but not implemented:

Allowing parallel translations on the same static key requires a restriction to the application. A single translation has to be used first to trigger reading the key descriptor into the cache before multiple translations can be allowed. The reason is that the tag management in the MMU and memory interface logic cause problems when multiple reads to the same address are simultaneously active. Although the design effort involved was not high, the extra verification effort did not justify the advantage.

A second proposal was considered following concerns about the performance when a server is connected to a storage device, such as IBM System StorageTM DS8000. As a server has many critical connections, it is not possible to rely on the use of static keys for the storage communication. However a fast reaction of the server is necessary to allow many parallel connections to the storage device. Therefore we proposed the partitioning of the cache holding the key descriptors, based on several inputs. As inputs, the LPAR, a hash from the key, information contained in a page descriptor register, and in the key descriptor itself, are assembled in a configurable way into a partition identifier. As the cache is fully associative, the partition identifier needs to be applied only during allocation of a new cache line, including cast-out of a previous line. The decision not to implement this feature was primarily based on the required support by the hypervisor, drivers, and applications. In particular, the latter are already complex, and further specialization to one hardware interface is not justifiable.

Acknowledgement

Many people contributed to the work presented here. With high respect for the work of all others only two architects, David Craddock, Poughkeepsie, and Thomas Schlipf, Böblingen, can be named here.

References

1. IBTA: Infiniband architecture specification 1.1 (November 2002)
2. Schlipf, T., Helms, M., Ruf, J., Klein, M., Dorsch, R., Hoppe, B., Lipponer, W., Boekholt, S., Röwer, T., Walz, M., Junghans, S.: Design and verification of the IBM System z10 I/O chips. IBM Journal on Research and Development **53**(1) (January 2009) in press, available at <http://www.research.ibm.com/journal/rd/>.
3. Georgiou, C.J., Salapura, V., Denneau, M.: A Programmable, Scalable Platform for Next-Generation Networking. In: Network processor design: issues and practices. Morgan Kaufmann (2003) 11–28
4. Hoefer, T., Rehm, W.: A communication model for small messages with infiniband. In: PARS Mitteilungen. Volume 22. (2005) 32–41

Grid Workflow Recovery as Dynamic Constraint Satisfaction Problem

Stanimir Dragiev and Jörg Schneider

TU Berlin, Germany
{stanio,komm}@cs.tu-berlin.de

Abstract. With service level agreements (SLAs) the Grid broker guarantees to finish the Grid jobs by a given deadline. There are a number of approaches, to plan reservations to fulfil these deadline requirements and to handle currently running jobs in the case of a resource failure. However, there is a lack of strategies to handle the already planned but not yet started jobs. These jobs will be most likely also affected by the resource failure and can be remapped to other resources well in advance. Complex Grid jobs (Grid workflows) consisting of multiple sub-jobs introduce a higher complexity to determine a remapping saving as much Grid jobs as possible. In this paper a recovery scheme for Grid workflows using a dynamic constraint solver is presented and the gain in the number of saved Grid jobs is evaluated using extensive simulations.

1 Introduction

Grid computing opens lots of new perspectives for the application of the traditional information technologies. They range from on-demand high performance computational power for home usage to realisation of complex scientific applications involving multiple enterprises from different cultures, background, locations. At the same time, the new usage possibilities and the nature of the grid give rise to new challenges. We adopt a common view of the grid as a physically wide distributed collection of resources of different types which are spread across administrative and state boundaries.

The idea to reproduce the organisational structure of the resources is the basis for the design [1] of the Virtual Resource Manager (VRM). It consists roughly of two layers: An administrative domain controller (ADC) is concerned with co-allocation and other grid-specific functions and allows for nested administrative domains (AD); Every managed resource is registered with the ADC via an active interface (AI) which addresses the heterogeneity by implementing the capabilities needed for participation in the grid and thus giving a unified view for all resources.

The central question for the ADC is how to provide reliable information on the ability to process given reservation request, as well as to give QoS guarantees and to adhere to the negotiated service level agreements (SLA). To this end, the architecture of the ADC encompasses three stages of scheduling: Triggered by an incoming request for reservation, an online scheduler uses heuristics to produce a valid schedule which meets the needs of the reservation, without breaking already accepted SLAs. As a next stage, a background scheduler optimises the effective schedule towards the policy of the administrative entity. The third stage is concerned with the handling of resource shortcomings resulting from failures.

Dealing with failures has two aspects: support for jobs directly affected by a resource crash and support for jobs scheduled to start on the failed resource in the future. The first domain includes techniques like checkpointing, migration and restart, and enjoy much attention from researchers. However, our work is concerned with the second aspect of recovery scheduling, i.e., the remapping of all planned, but not yet started jobs affected by the outage. A fundamental problem in this domain is the uncertainty about the downtime of a given resource. In general, there are no reliable means to determine the nature of a failure at the time it happens; we can merely observe the result of it: the non-availability of the resource.

Under- or overestimation of the downtime leads to poor quality of the future schedules and thus to limited satisfaction of the administrative policy [2]. Hence, mechanisms are needed to adapt the reactions to the downtime.

Recovery means, after all, to choose a valid schedule amongst large number of possible ones, whereby the choice is driven by feasibility criteria, job requirements and resource management policies. The domain concerned with precisely this kind of search problems is Constraint programming. Here, we propose to restate the recovery scheduling in terms of Constraint satisfaction problem (CSP) and benefit from approaches used in similar situations in other research areas. We are particularly interested in the concept of CSP with changing constraints over time, Dynamic CSP (DCSP), and the idea of maintaining similar consecutive solutions.

Grid applications usually comprise several phases – data collection, processing, dissemination of results. These phases are realised by sub-jobs, which interact with each other. In other words, there are temporal and spatial dependencies between the single jobs, as well as QoS requirements on the resources responsible for running the jobs. The notion of Grid workflow [3] helps to consider all parts of a large application as a whole and to model the requirements in a consistent way. Important property of Grid workflows is that all sub-jobs contribute an essential part to the accomplishment of the final goal of the workflow, i.e. a workflow is successfully completed only if no jobs fail.

In the following section, we comment known efforts to deal with scheduling and recovery in the grid, as well as CSP techniques for similar problems. After discussing our approach to determine a recovery schedule, we show the evaluation results.

2 Related Work

The simple and robust approaches of managing resources, like providing only batch processing, are still widely used in practice, including MOAB [4], LSF [5], PBS [6], LoadLeveler [7], Globus Toolkit [8]. Despite their maturity, these techniques are not fully capable to ensure the QoS negotiated in service level agreements. The required advance is provided by middleware solutions incorporated in grid management systems like VRM [1] or by advanced local scheduler like CCS[9] or MAUI[10].

In [11] requirements for failure handling in the grid are defined. The emphasis is on the need of application context dependent recovery and distinct recovery on task and workflow level. The task level related techniques include *checkpointing*, *migration*, *restart*, and *replication*. All consideration are concerned with directly affected jobs.

The automatic recovery of future jobs affected by a failure is the topic of [12]. The authors propose an *adaptive, load based* downtime independent algorithm. The basic idea behind it is the introduction of a *remapping interval* which is calculated based on the current load situation. The resource is then assumed unavailable for this interval only and all jobs scheduled on the resource for this time are remapped. At the end of the interval the same procedure is repeated until the resource is up again.

The remapping interval computation is based on the average incoming load per timeslot and currently booked load for every particular timeslot. Taken at a particular timeslot, their sum serves as estimation of the expected load in it. The remapping interval is determined as the timeslot for which the expected load falls below given level. This approach handles only independent jobs and is extended in this paper for Grid workflows.

The well studied theory of Constraint satisfaction problems (CSP) can provide means for solving the problem of finding a valid schedule. However, a single CSP depicts the situation at a particular time. When resource availability changes, the constraints change, and this can

be seen as new CSP instance. For handling such problems the notion of Dynamic constraint satisfaction problem was introduced (DCSP) [13].

The combination of Job shop scheduling problem and Minimal perturbation problem is the topic of [14]. It is based on the idea to maximize similarity between the broken and the new schedule.

One way of maintaining minimal changes in solutions of consecutive CSPs is to produce schedules which are expected to remain valid after changes in the CSP [15]. For this approach to work properly it is assumed that certain types of *expected* changes are known which is limitedly applicable for environment with unexpected resource failures.

An alternative idea is proposed in [16] – to take the previous, usually invalidated by constraint changes CSP solution as basis for the search of a new one. The algorithm works with two subsets of variables – the ones allowed to change and the fixed ones. The approach maximizes the similarity only with respect to the set of fixed ones.

Several general purpose approaches which deal with over-constrained problems are discussed in [17], and this is, in general, the challenge in recovery scheduling. The guide introduces the notions of Fuzzy CSP, Probabilistic CSP, Weighted CSP Partial CSP (PCSP), constraint hierarchies and *higher-order constraints* (in other works referred to as reified constraints) . The latter allow for specifying predicates over constraints, like in c_i or *true*. They seem to be the best supported ones in existing frameworks, for example, in the constraint solvers built upon Gecode [18] which we employ in our work.

3 Approach

The recovery scheduling is triggered by a resource failure which possibly makes the existing schedule infeasible. Roughly, the proposed algorithm does the following: determine a *remapping interval*, compute a *horizon* (the maximum considered time for the remapping of jobs), transform the current physical limitations and workflow requirements to constraints, pass the resulting CSP to a constraint solver, implement the found solution as new schedule.

3.1 Remapping interval

The computation of the remapping interval is done according to [12]. The adaptation effect of the remapping interval is made dependent on the current load situation. Effectively, it leads to shorter intervals in heavier loads and longer intervals for more relaxed situations. For higher load, this forces the algorithm to eagerly search for possibilities to recover future jobs.

Once the remapping interval is computed it determines the set of jobs we consider for remapping: the ones scheduled to start in the remapping interval – both on broken and intact resources. Figure 1 shows a synthetic schedule and the sorts of job from the recovery scheduler point of view. We consider the not completed jobs on the failed resource terminated. As consequence, their entire workflows are dropped from the schedule (workflows *A* and *B* in the example). The jobs scheduled to start in the remapping interval on the failed resource need to be relocated in order to be able to run (C_1 , D_3). By allowing a reordering of other jobs starting in the remapping interval (D_2 , E_1 , C_2) we gain additional possibilities to obtain optimal schedule.

3.2 Schedule Representation

The considered jobs, their dependencies, the timeslots they run on, and the available resources define the space for the search for new schedule. This search space have to be

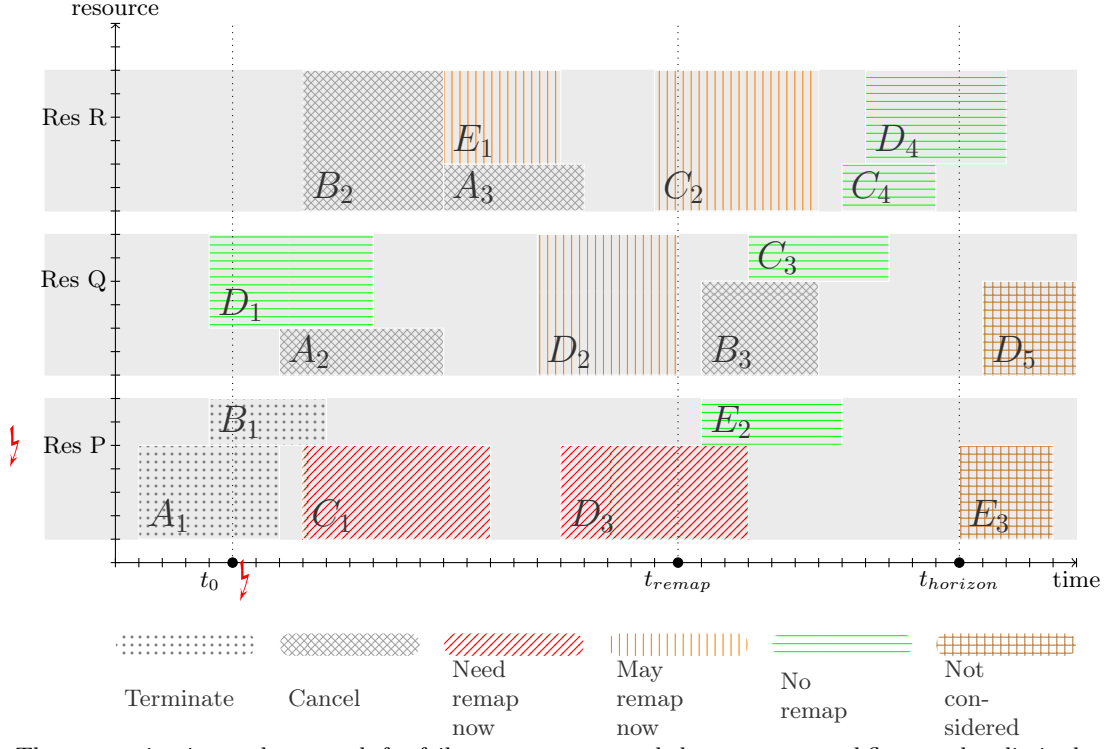


Fig. 1. The remapping interval approach for failure recovery extended to support workflows and to limit the search space size. Workflows: A, B, C, D, E . Resources: P, Q and R ; Resource P crashes in t_0 . The approach extension cancels the jobs depending on terminated ones, and doesn't consider the ones beyond $t_{horizon}$.

represented in a form suitable for expressing constraints on it. We put it in terms of an integer problem:

Definition 1. Let $T := \#\{\text{timeslots}\}$, $R := \#\{\text{resources}\}$, $J := \#\{\text{jobs}\}$. Then, $S_{T \times R \times J}$ is 3D integer matrix, whereas $s_{t,r,j} = x \iff$ "in time slot t job j uses x of resource r "; with $s_{t,r,j} \in \{0, J_j.\text{demand}\}$

In this representation, we can constrain the variables $s_{t,r,j}$ and employ a constraint solver to find a valid solution.

3.3 Constraints

The next task is to elaborate constraints which express the capacity limits of the local resources and the requirements of the accepted reservations. These are basically:

Resource capacity: The sum of consumed entities of a resource cannot exceed the resource capacity.

Enough resources in timeslot: In every particular timeslot in which a job runs, it has enough resources to run properly.

Not split: Every job runs on one resource only.

Workflow atomicity: A workflow depends on all its jobs, thus either all jobs can be scheduled, or no job at all. The latter removes needless load from the system.

Temporal relations: If job j_A is explicitly specified to run after a j_B , the jobs should run in the given order.

Parallel jobs: Jobs specified to be parallel need to have resources to run in the same timeslots as long as the longest job requires.

Enough time: Jobs have to be assigned to a resource for enough timeslots.

Types match: Jobs are assigned to resources capable of running them.

Data dependencies: The data prerequisites of a job need to be satisfied, i.e. if job j_B relies on input by job j_A , they have to run in order j_A and then j_B , whereas if they do not run on the same resource, a transfer job J_{AB} needs to be scheduled between them.

Connected assignment: The timeslots a resource is assigned to a job have to be consecutive ones.

Workflow times: All jobs start after the start time of the workflow and end before the workflow end time.

The formal definition of the constraints is given in First order logic (FOL). As an example, consider the three constraints of figure 2.

Resource capacity	$\forall t \forall r (\sum_{j=0}^j s_{t,r,j} \leq R_r.cap(t))$
Workflow atomicity	$\forall j \forall j' (wfl(j, j') \rightarrow (\sum_{t,r} s_{t,r,j} = 0 \rightarrow \sum_{t,r} s_{t,r,j'} = 0))$
Connected assignment	$\forall j \forall r (\forall t_r \forall t_f (rise(t_r, j, r) < fall(t_f, j, r))),$ $rise(t_x, r, j) := t_x = -1 \vee (0 \leq t_x < T - 1 \wedge s_{t_x, r, j} = 0 \wedge s_{t_x+1, r, j} > 0),$ $fall(t_x, r, j) := t_x = T - 1 \vee (0 \leq t_x < T - 1 \wedge s_{t_x, r, j} > 0 \wedge s_{t_x+1, r, j} = 0).$

Fig. 2. FOL representation of constraints. Examples.

3.4 Over-constrained problem

Obviously there are cases, a failure not only invalidates the current schedule but tightens the resource constraints so much that there is no schedule in which all accepted reservations fit. For such over-constrained problems, *reification* allows to relax selected constraints and to find the best available solution. Reification means to break a constraint, in order to make the problem solvable.

This makes sense for constraints which do not represent physical limitations, i.e. only for the ones which determine whether a job runs at all. In our model, it is not feasible to reify, for example *Resource capacity*, or *Workflow times*. Both would lead to a schedule which is infeasible and thus to no benefit from the recovery.

In contrast, the more jobs are dropped from the schedule, the easier to find valid assignment for the rest. In other words, if we allow to break constraints like *Enough resources in timeslot* we can adjust the load to the available resources. Effectively, this ensures the existence of solution and passes the responsibility for determining the optimal schedule to the objective function

3.5 Optimisation Objective

The notion of *optimal schedule* implies an objective for the search process. In order to counteract to the reification, we want as the primary goal to maximize the scheduled load¹: $W(S) = \sum_{t,r,j} s_{t,r,j}$, with t spanning the indexes of the considered timeslots, r – the resource indexes, and j – the job indexes. As a secondary objective we impose on the solution the condition to remain as close as possible to the broken one: $D(S, S') = \sum_{t,r,j} |s_{t,r,j} - s'_{t,r,j}|$. The latter is a similarity measure for schedules and constitutes the connection between consecutive CSP solutions and eventually utilises the quality of the old schedule.

¹ this may be seen as a measure for the earnings of the resource provider

4 Evaluation

To evaluate our approach, the recovery scheme was implemented as a module of the grid management system VRM [1]. We simulated a Grid with 8 computational resources, providing between 32 and 512 CPUs each. Sequences of randomly generated workflows were scheduled to simulate the normal operation. Then, one of the resources was assumed to fail, to start the recovery mechanism.

We simulated a variety of parameters which we expected to impact the performance: load, job length, time available for recovery, number of jobs per workflow. The experiments in Figure 3 already show the significant advance of the recovery mechanism compared to naively dropping all affected future jobs and the workflows they belong to. In addition, the approach is often able to preserve more than 80% of the pre-downtime load.

All pictures show on the y-axis the preserved load normalized to the total load before the resource failure, with respect to varying parameters on the x-axis. The upper line is the total pre-downtime load itself – constantly one. The lower line is the load remaining in the schedule after the failure without any recovery mechanism. The middle line represents the proposed recovery scheme using DCSP.

The four experiments depicted in Figure 3 show that the performance of the recovery scheme is independent of the system parameter. By changing the interarrival time we modelled highly utilized grids (low interarrival time) and lower utilized Grids. In the upper right diagram we modified the complexity of the Grid workflows to be scheduled, which obviously has an impact at the loss if no recovery is done but hardly any impact on the recovery scheme. In the lower row we modified the size of the jobs in the resource and time dimension.

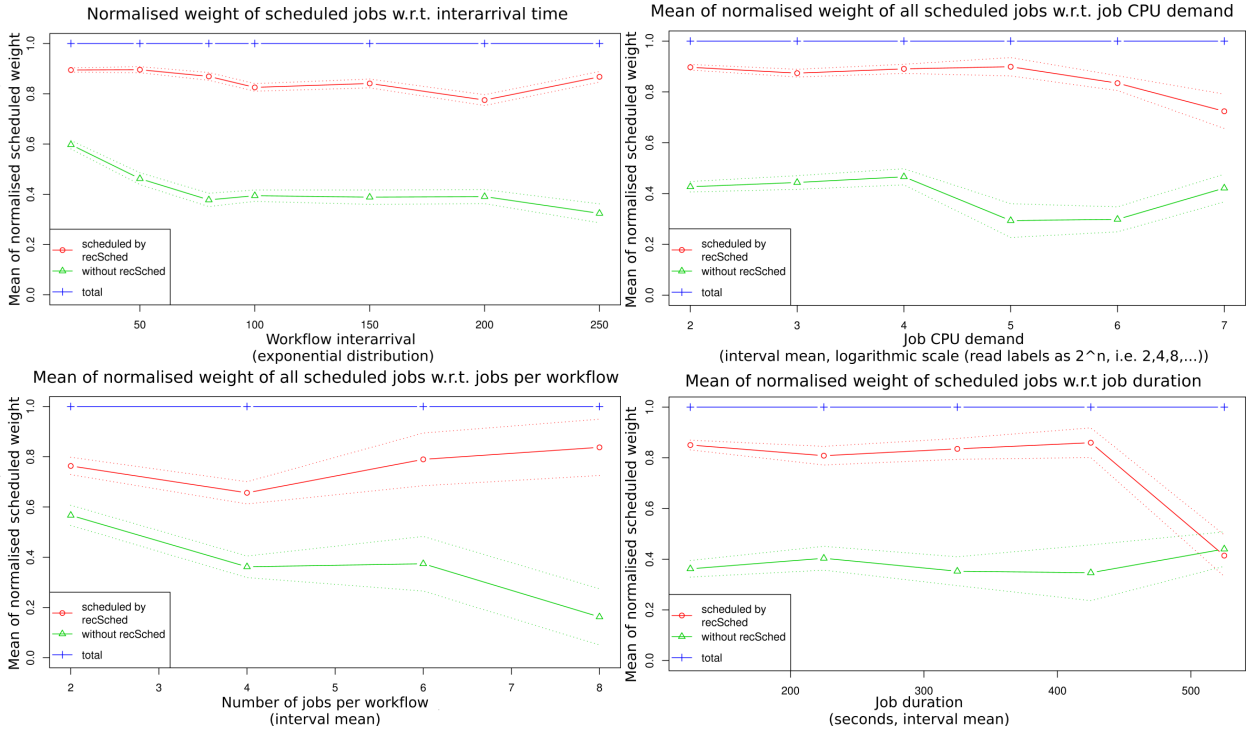


Fig. 3. Simulation results from the prototype implementation show the pre-failure load, the load after applying our approach and the load without recovery. Dotted lines show the confidence interval to the 99% level around the mean from multiple measurements.

5 Summary and outlook

We presented an approach to recover Grid workflows after resource failure. It focuses on the future jobs scheduled on the failed resource. Furthermore, we propose to employ a constraint solver for the search for optimal schedule. To this end, a formal representation of a schedule as an integer matrix is elaborated, as well as a constraint system which reflects the feasibility requirements for a schedule. The objective function for the optimisation includes maintaining similarity to previous solutions.

Future research will go in multiple directions: An alternative schedule representation can bring rigorous improvements in the complexity of constraint description and in the computational complexity. Study of the internal parameters of the approach, including the use of heuristics in the search phase, can lead to performance enhancements. Furthermore, formulation of the constraint in a portable way will allow for use of different constraint solvers and the strategies they implement.

References

1. Burchard, L.O., Hovestadt, M., Kao, O., Keller, A., Linnert, B.: The virtual resource manager: An architecture for SLA-aware resource management. In: 4th Intl. IEEE/ACM Intl. Symposium on Cluster Computing and the Grid (CCGrid) 2004, Chicago, USA. (2004) 126 – 133
2. Burchard, L.-O., Linnert, B.: Failure Recovery in Distributed Environments with Advance Reservation Management Systems. In: 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM), Davis, USA. Volume 3278 of Lecture Notes in Computer Science (LNCS), Springer (2004) 112–123
3. Hollingsworth, D.: The workflow reference model version 1.1. Technical Report WPMC-TC-1003, Workflow Management Coalition (1995)
4. MOAB Team: MOAB workload manager. on-line (2009) <http://www.clusterresources.com/products/moab-cluster-suite>.
5. Platform Computing Inc.: Platform LSF. on-line (2009) <http://www.platform.com/products/LSFfamily>.
6. Altair Grid Technologies: PBS Pro administrator guide 5.4. on-line (2004)
7. IBM: Tivoli workload scheduler LoadLeveler. on-line (2009) <http://ibm.com/systems/clusters/software/loadleveler>.
8. Foster, I., Kesselman, C.: Globus: A metacomputing infrastructure toolkit. International Journal of High Performance Computing Applications **11**(2) (1997) 115
9. CCS Team: CCS: Computing Center Software. on-line (2009) <https://www.openccs.eu/core/>.
10. Brett Bode, David M. Halstead, R.K., Lei, Z.: The portable batch scheduler and the maui scheduler on linux clusters. In: Proceedings of the 4th Annual Linux Showcase & Conference, USENIX Association (2000)
11. Hwang, S., Kesselman, C.: Grid workflow: A flexible failure handling framework for the grid. Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC03) (2003)
12. Burchard, L.O., Linnert, B., Schneider, J.: A distributed load-based failure recovery mechanism for advance reservation environments. In: Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on. Volume 2. (2005) 1071–1078
13. Dechter, R., Dechter, A.: Belief maintenance in dynamic constraint networks. In Smith, R.G., Mitchell, T.M., eds.: Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, Minnesota, Morgan Kaufmann (1988) 37–42
14. Ran, Y., Roos, N., van den Herik, J.: Methods for repair based scheduling. Proceedings of the 21th workshop of the UK PLANNING and SCHEDULING Special Interest Group, PLANSIG 2002 (2002) 79–86 (2002)
15. Wallace, R.J., Freuder, E.C.: Stable solutions for dynamic constraint satisfaction problems. In Maher, M.J., Puget, J.F., eds.: Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26–30, 1998, Proceedings. Volume 1520 of Lecture Notes in Computer Science., Springer (1998) 447–461
16. Verfaillie, G., Schiex, T.: Solution reuse in dynamic constraint satisfaction problems. In: Proceedings of the National Conference on Artificial Intelligence (AAAI 94). (1994) 307–312
17. Bartak, R.: On-line guide to constraint programming, 1st edition; 1st part: Constraint satisfaction. on-line (2009) <http://ktiml.mff.cuni.cz/~bartak/constraints>.
18. Gecode Team: Gecode: Generic constraint development environment. on-line (2009) <http://www.gecode.org>.

A Dependable Middleware for Enhancing the Fault Tolerance of Distributed Computations in Grid Environments

André Luckow

Institute of Computer Science
Potsdam University
luckow@cs.uni-potsdam.de

Abstract. The development, deployment and execution of Grid applications is a challenging task. Grids are by definition dynamic, heterogeneous and failure-prone environments. Especially for long-running scientific applications high failure rates represent a major barrier for the adoption of Grids. In this paper, we present the *Migol* middleware. Migol addresses the fault tolerance of long-running scientific applications. It supports applications in performing complex tasks such as resource allocation, monitoring, checkpointing and, if necessary, recoveries. We provide a general overview of Migol and extensively discuss important design issues, such as failure detection and handling in highly distributed systems, that arose during the development of the Migol middleware.

Key words: Grid Computing, Fault Tolerance, MPI

1 Introduction

Grid computing envisions the sharing of compute, storage, network and software resources across multi-institutional virtual organizations (VOs) to provide more effective solutions for important scientific, engineering and business problems [5]. With the advancing penetration of Grid infrastructures, the fault tolerance of these environments becomes more important. Different studies show that in large computational Grids resources become unavailable at very high rates: In the Grid 5000 it was found that the mean time to failure for nodes is 12 minutes at Grid-level, five hours at cluster-level and two days on node-level [9]. An analysis of high performance systems at the Los Alamos National Laboratory conducted by Schroeder/Gibson found similar high failure rates [20]. Other studies investigated the failure rates of Grid jobs and also showed significant deficits: Depending on the scenario 10-40 % of all submitted jobs failed [11, 14]. Although these analyses have been conducted in different environments with different hardware/software configurations and measurement methods, it is obvious that reliability is not an intrinsic property of Grid environments. Efficient support for fault tolerance is therefore a key requirement for Grid infrastructures.

In particular for long-running applications high failure rates are a major concern. Many scientific applications require to run for days or weeks. For example, a simulation of gamma-ray bursts, an astrophysical phenomenon, requires over 100 days of runtime on a one PFlop/s machine [18]. That means that usually multiple recoveries are necessary to complete a single application run. A particular concern is the inability of existing parallel programming systems, such as the Message Passing Interface (MPI), to handle failures gracefully. A single node failure usually leads to the abort of the entire application. For highly distributed and failure-prone Grid environments additional measures are necessary to ensure the fault tolerant execution of MPI applications in Grid environments.

The capability of a system to continue its operation despite the presence of faults is defined as *fault tolerance*. To handle the increasing complexity of large-scale, distributed Grids, systems should ideally resolve failures automatically without user involvement. This

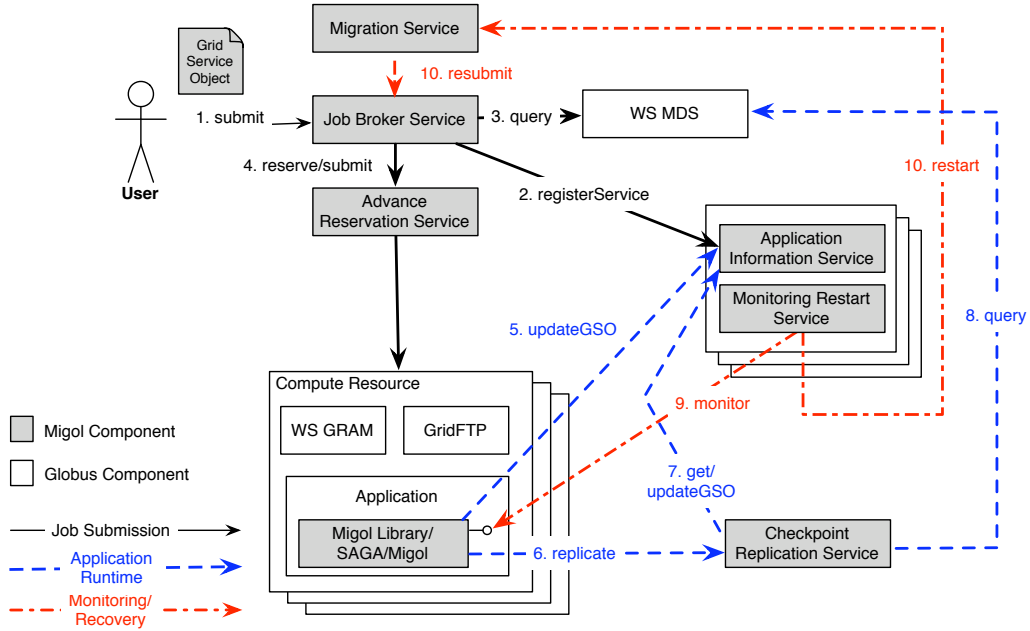


Fig. 1. Migol Service Architecture: Migol provides an autonomic service environment for supporting the fault tolerance of Grid applications. Applications that are managed by Migol are transparently monitored and automatically recovered in case of failures.

capability is often referred to as *self-healing* [10]. In this paper, we present the *Migol* framework [16], a dependable and autonomic environment that follows the autonomic computing principles to support the fault tolerance of long-running Grid applications. Migol addresses the complete lifecycle of Grid applications – from the discovery of a resource, the handling of data transfers and checkpoints, to the management of the application runs. The most important feature of Migol is the ability to automatically detect and handle different types of failures.

In this paper, we investigate important design issues that arose during the development of an autonomic infrastructure for supporting the reliable execution of Grid applications. After an overview of the Migol framework in section 2, typical failure modes that applications can encounter during their runtime in Grid environments are analyzed in section 3. Further, we highlight important design issues and approaches that Migol uses to address such failure modes. Section 4 describes the pre-cautions taken to ensure the fault tolerance of the infrastructure. Experiences with Migol and several Grid applications are presented in section 5. We analyze related work in section 6.

2 Migol Overview

Migol guarantees the correct and reliable execution of Grid applications even in the presence of failures. The architecture of Migol follows the Open Grid Services Architecture (OGSA) [4] vision of a service-oriented Grid comprising of a set of loosely coupled services. Services are modular components that expose the capabilities of a resource via a standardized WSRF interface [1]. The framework is based on the Globus Toolkit 4. Existing services, components and libraries, especially those of the underlying Globus framework are re-used if possible. Figure 1 shows the Migol architecture and the interactions between the different services. The used acronyms are given in table 1.

The fundamental metadata model of Migol is the *Grid Service Object (GSO)* schema. A GSO describes a Grid application and contains all information that is required for the

GSO	Grid Service Object
AIS	Application Information Service
MRS	Monitoring Restart Service
MS	Migration Service
JBS	Job Broker Service
ARS	Advance Reservation Service
CRS	Checkpoint Replication Service
WS MDS	Web Service Monitoring and Discovery Service
WS GRAM	Web Service Grid Resource Allocation Manager

Table 1. Migol Acronyms

management of the application run. All GSOs are stored in the *Application Information Service (AIS)*, the central registry service of Migol. To avoid a single point of failure, the AIS is replicated using a Grid aware replication protocol [17].

Migol manages the complete application lifecycle. Applications are started via the *Job Broker Service (JBS)* (step 1, figure 1). To ensure that the user can later locate its application, the JBS initially registers the GSO of the application at the AIS (step 2). Resource discovery is performed through the Monitoring and Discovery Service (MDS) [19] (step 3), which aggregates resource information from different services, e.g. another MDS and the Network Weather Service [21]. Available resources are matched by the JBS with the application requirements. Depending on the configured scheduling strategy, the JBS selects an appropriate resource from the candidate list. For the execution of applications on Grid resources, the JBS uses the Globus GRAM [3]. The *Advance Reservation Service (ARS)* supports resource reservation on top of GRAM and is used for managing jobs on reservation-aware resources (step 4).

For recovery, Migol relies on application-level checkpointing, i.e. applications have to be written to accommodate checkpointing and restarting. The checkpoint metadata is maintained in the GSO at the AIS. It must be updated by the application each time a new checkpoint is written. For this purpose, the Migol library, which can be accessed via different APIs, is used (step 5). To ensure the availability of checkpoint files even in the case of a failure, checkpoints can be replicated to an alternative location using the *Checkpoint Replication Service (CRS)* (step 6-8).

Migol provides several mechanisms for supporting the fault tolerance of Grid applications. To detect failures, the *Monitoring and Restart Service (MRS)* monitors periodically all applications registered with the AIS (step 9). In case the MRS discovers a failed application, it initiates a recovery using the *Migration Service (MS)* (step 10).

A Migol application traverses various states during its lifetime. Figure 2 summarizes the underlying state model. The JBS assigns initially the state *unsubmitted* to the application. After completion of the selection procedure and the successful submission of the application to at least one LRMS queue the state is changed to *pending*. After the successful startup, the application state is updated to *active*. This state is set by the application itself respectively by the Migol library. If the application cannot be submitted to any resource, the JBS will terminate the procedure after a threshold of tries – the state is then set to *inactive*.

Once the application becomes active, the MRS is responsible for monitoring. If the application terminates abnormally, the MRS detects this failure and adjusts the application state to *inactive*. The MRS then attempts to recover the application using the MS/JBS. The state changes from *inactive* to *migrating*. In case of a successful submission the states *unsubmitted*, *pending* and *active* are traversed. A failure can occur during any phase (except the state *done*) – the application state is then set to *inactive*.

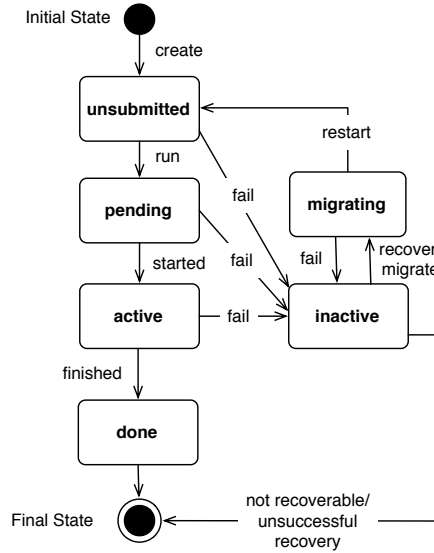


Fig. 2. Migol's Application State Model: During its lifecycle a Grid application runs through different states. In contrast to state models used by most LRMSs and the Globus Toolkit, the Migol state model considers possible failures and recoveries.

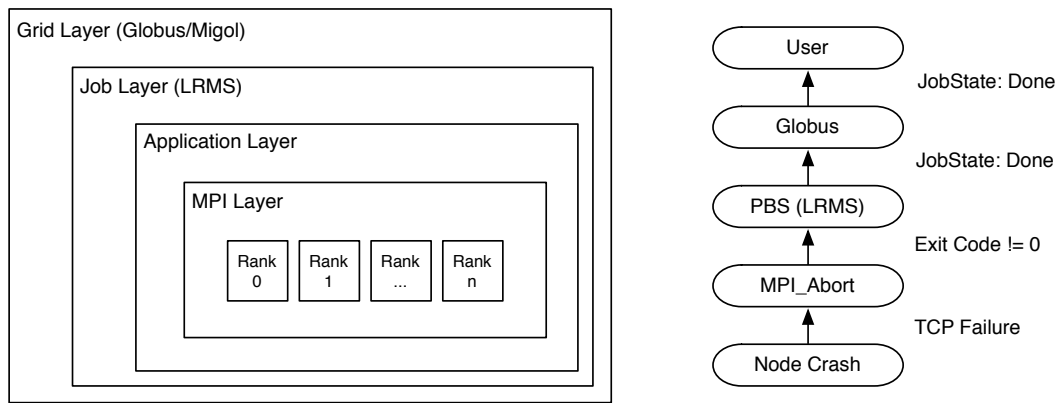


Fig. 3. Error Propagation: The execution of a Grid job involves multiple software layers. Failures should be handled at the lowest possible layer. If a failure cannot be masked, it must be propagated to an upper layer, which can then attempt to resolve it.

3 Application Fault Tolerance

To achieve a dependable system, fault tolerance must be introduced into the system design from the beginning. Failure detection is the necessary pre-requisite to provide fault tolerance. Both aspects are described in the following.

3.1 Failure Detection

Failure detection within distributed systems consisting of multiple layers from hardware to software is difficult. Since failures must be propagated through multiple layers, the failure detection latency is increased, the failure behavior becomes more unpredictable and the diagnosis of fault causes is complicated.

Figure 3 exemplifies a typical error propagation chain of a Grid job running an MPI application. Several software layers are involved during the execution of a job:

- The job is started using the Grid middleware, i. e. Migol/Globus (Grid layer).
- The LRMS (job layer) is responsible for running the MPI application on the local resource.

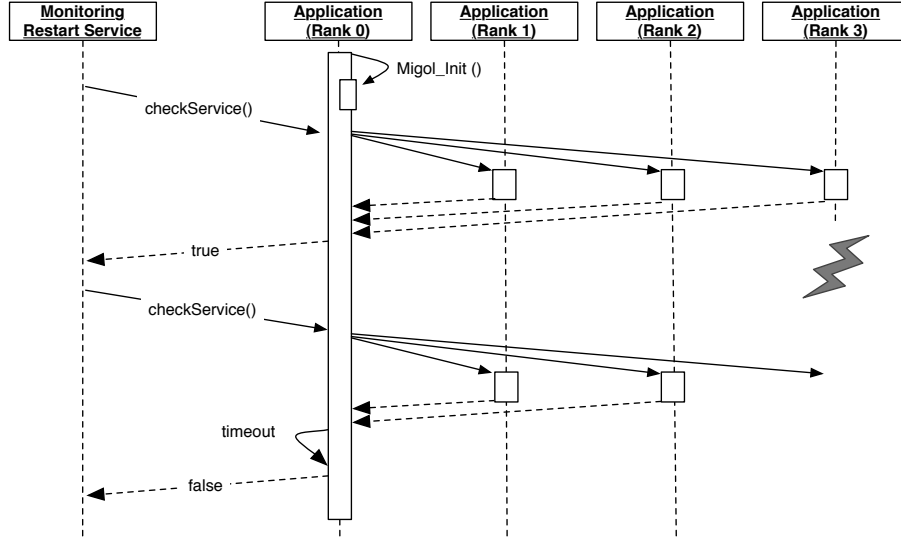


Fig. 4. Migol Library Failure Detection: During the call to `Migol_Init`, the Migol library initializes a HTTP server, which is used to directly propagate errors to the Migol infrastructure. The MRS periodically monitors the application. With each monitoring request, the Migol library verifies the state of all MPI processes.

- The `mpiexec` script launched by the LRMS spawns the MPI application processes (rank 0...n) and monitors them during the runtime (application and MPI layer).

To enable an automatic failure handling, error conditions must be precisely described and accurately propagated to the middleware. Unfortunately, the failure behavior of MPI based applications is usually very inconsistent [7]. In the best case the failure of an MPI process causes the termination of the application. In many cases however, such a failure leads to a hanging MPI application. Since LRMSs usually solely monitor the operating system process, i.e. `mpiexec`, they are not able to properly detect such errors. Nevertheless, the failure of a single node should not cause a failure of the entire application. Thus, it is desirable to recover from partial failures, such as a node crash, on the lowest possible layer, i.e. from within the MPI application. Without this ability, additional measures are necessary to run MPI applications within highly distributed and error-prone Grid environments.

Often, errors are not forwarded correctly from the application to the LRMS and the Grid middleware. If e.g. an MPI application terminates abnormally, the non-zero return code is not interpreted by PBS/PBSPRO. Figure 3 illustrates this deficit: PBS does not consider the return value of the application and sets the job state to *done*. Thus, on Grid-level it cannot be differentiated between a normal and abnormal termination of the application. To address these issues, the Migol library provides 1) a failure detector for parallel MPI applications and 2) a monitoring Web service that allows the Grid middleware to directly diagnose applications and to detect application-level errors. Further, the Migol library allows the implementation of a custom, application-specific failure detection module.

Figure 4 illustrates the failure detection process. The Migol library uses a lightweight HTTP server to implement the **Monitorable** Web service. The MRS uses this service to ensure that the application is alive. If a monitoring request is received, the configured failure detector is triggered. Using a direct link between the Grid middleware and the application ensures that failures are timely and correctly detected and avoids the error-prone failure detection chain through the LRMS and Globus.

After reception of a monitoring request via the **Monitorable** service, the MPI failure detector verifies all MPI processes using a keep-alive message. For this purpose, a separate thread at each MPI process is utilized. Again, suitable timeouts are critical to distinguish between a slowdown and a failure. Timeout intervals can be configured by the application.

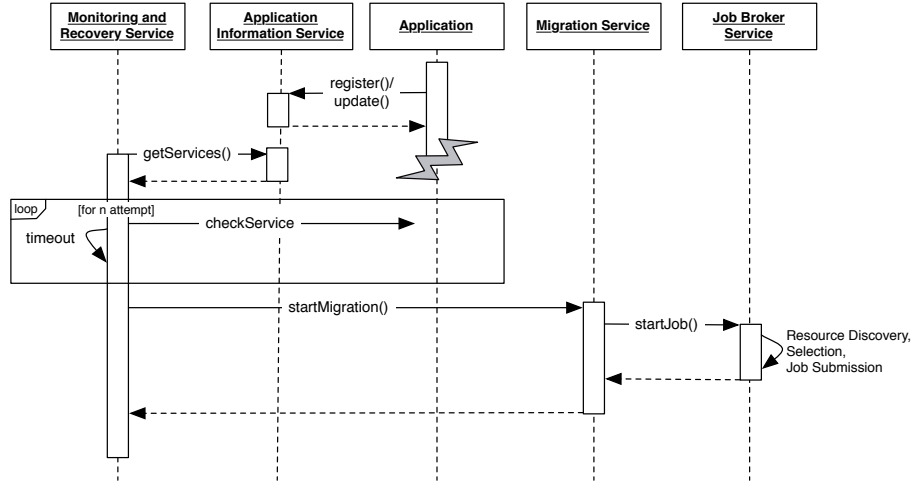


Fig. 5. Failure Detection and Recovery: Applications must register a GSO at the AIS. The MRS periodically monitors all applications. After a threshold of failed monitoring attempts, a recovery is initiated through the MS and JBS.

If an error is detected, it is directly propagated to the MRS, which will then attempt to resolve the failure. The Migol MPI failure detector can be seamlessly integrated with any MPI application using the Migol library.

For large MPI applications running on hundreds of nodes, this solution might impose some overhead. Thus, the Migol library allows the implementation of a custom failure detection module. An application can e.g. use the progress of an application-specific state variable, such as the current iteration, as failure indicator. For example, a linear solver could trigger a failure if a convergence has not been reached for a certain time. As with the MPI failure detector, the custom failure detector module can be triggered by the MRS using the `Monitorable` service.

3.2 Failure Handling

Figure 5 illustrates the failure handling procedure. For failure detection Migol relies as described on the *Monitoring Restart Service (MRS)*, which periodically checks all services registered at the AIS. To distinguish between failures and slowdowns the MRS uses the following configurable parameters: the maximum response timeout, the retry interval and the number of retry attempts. To avoid false failure detection, high failure thresholds are used by default: a timeout of 60s, a retry interval of five minutes and three retry attempts.

A Grid provides sufficient resource redundancy, i. e. usually free resources exist that can be dynamically allocated in case of a failure. If an unreachable application is detected, a backward recovery is initiated by restarting the application on another resource using the *Migration Service (MS)*. Further, users, applications or local system administrators can also trigger migrations.

After initiation of a migration, the MS attempts to checkpoint the target application. If the application is not available, the most up-to-date and reachable checkpoint registered at the AIS is used. After checkpointing, the MS seeks to gracefully stop the application. The application is then restarted using the last stored checkpoint and the JBS. On the new resource, the application process is recreated and the state is restored from the checkpoint.

4 Infrastructure Fault Tolerance

All Migol services have been classified according to their criticality into less critical and critical services. Based on this classification, an overview of the fault tolerance techniques used for each service is given.

4.1 Less Critical Services

The criticality of a service depends to a great extent on the criticality of the state it maintains. A failure of a less critical service has only a small impact on the overall system, e.g. the loss of some resource states affecting only a few users. This definition applies to the following Migol services: the Advance Reservation Service, the Job Broker Service, the Migration Service and the Checkpoint Replication Service as well as to the Globus MDS.

Migol relies on different simple techniques to assure the fault tolerance of these services. All services are designed to support a recovery after a failure, i.e. they record critical state information on a stable storage. After a failure, services are able to restore their state and resume operation. Thus, clients can resolve transient service failures by simply re-executing their request. Further measures are necessary to cope with permanent failures. The architecture ensures that always multiple, redundant services with the same capability are deployed in a Grid. But, in contrast to actively replicated services, these replicas are not tightly synchronized. If a fail-over is necessary, clients must re-execute their request at another replica. Since these services only maintain a short-lived state, the performance penalty for re-execution is acceptable. The JBS e.g. only manages the startup of a job. In general, the startup time is very low compared to the overall runtime of Grid applications. Thus, the time necessary for re-executing a job submission in case of a failure during the job start is also very short.

While the Migol services provide fault tolerance support, clients must anticipate possible transient and permanent service failures. Clients must e.g. be able to re-execute their request at the same or in case of a permanent failure at another service replica. For every Migol service, redundant service replicas are deployed in a Grid. Every replica is registered at the AIS. Since the AIS is designed for high availability, it is ensured that clients can discover another service replica at any time. Both an automatic retry and a re-discovery mechanism is built into the AIS client library and the Migol library, i.e. the application is not required to perform these actions manually.

Another less critical service is the MDS. In case of a failure, the state of the MDS can be completely restored by re-collecting the resource information. Thus, no state persistence is required. The fault tolerance of the MDS is ensured by deploying multiple, hierarchically organized MDS in a Grid. Each GT4 container contains an MDS, which aggregates local resource information. In addition, sites and VOs usually run an MDS, which collects information from different local MDS. Since several MDS cache the same data, the MDS does not represent a single point of failure. The JBS and the CRS depend on the MDS for resource discovery and selection. Both services can be configured to query and cache information of multiple MDS. Since the MDS replicas are registered at the AIS, the JBS and the CRS can discover an alternative MDS if necessary.

4.2 Critical Services

The failure of a critical service has a major impact on the entire Grid environment, i.e. in general multiple services and Grid applications are affected by this failure. For Migol this definition applies to the Application Information Service and the Monitoring Restart Service. A failure of the AIS or the MRS has severe consequences:

- The AIS serves as central registry service. During the lifetime of an application, users and services can obtain the current location of an application by querying the AIS. If the AIS is not available, users can neither locate their application nor can applications access or modify the stored metadata. Among other Migol services, the Migration Service requires e. g. the application metadata to conduct an automatic recovery.
- The MRS must be able to obtain the monitoring endpoints of all Grid applications from the AIS. If the AIS fails, no monitoring is possible.
- If the MRS fails, no monitoring is performed. Thus, no application failures can be detected and no recoveries can be performed.

The architecture must ensure that these services are continuously available. To achieve this, the AIS and MRS are replicated across multiple Grid nodes. Since the AIS stores stateful data, which is both read and updated, ensuring data consistency across multiple nodes is difficult. To avoid that queries to different service replicas return ambiguous results, a strongly consistent replication semantic is required. Hence, it must be ensured that all update requests are executed at all AIS replicas in the same order. For this purpose, the AIS relies on the *Web Service Ring Replication Protocol (WS RRP)* [17], an active replication protocol especially designed for Grid environments.

5 Experiences with Migol

Migol has been successfully integrated and tested with different Grid applications, e. g. with the n-body code AMIGA, the Cactus framework and a Replica-Exchange framework for studying the Hepatitis-C virus [15]. These applications and the Migol infrastructure have been successfully deployed to several production Grid environments: the D-Grid, the LONI Grid, the TeraGrid and the Amazon EC2 infrastructure [15–17]. In the following a long-term experiment that studies the robustness and effectiveness of Migol is described.

In this experiment the Cellular Automaton, a MPI-based mesh computation, is setup to run for an indefinite amount of time. As resources the three Grid clusters Einstein, Uranus and Highland of the Postdam University are used. The Migol infrastructure is deployed on three dedicated infrastructure nodes. Table 2 summarizes the results.

Runtime	53 days 23 h
Number of job failures	83
Number of AIS reconfigurations	4
Mean Time To Failure	14 h 26 min
Mean Time To Repair	4 min 11 s (automatic) 1 h 7 min (w/ manual)
Manual Recoveries	3

Table 2. Migol Long-Time Experiment

In total, the application run for 53 day and 23 hours. Since the used clusters are configured with a maximum walltime of 24 hours at least one recovery per day was necessary. In total, the application recovered 83 times, which is equivalent to a mean time to failure of about 14.5 hours. The recovery time for automatic recoveries averaged to 4 min 11 sec. In three cases a manual recovery was necessary; thus, the total mean time to repair amounted to 1 h 7 min. These three case were caused by some rare failure modes:

- Due to a reboot caused by the installation of kernel security patches, the AIS database was completely wiped. To avoid such an issue in the future, a mechanisms for clients to re-register a GSO was introduced.

- An unsynchronized clocks on a cluster node rendered delegated Grid certificates invalid. Thus, the application was not able to interact with Migol which led to its failure. The JBS re-scheduling mechanism was enhanced to trigger a re-scheduling in such cases.
- Due to an overload of the front end node (caused by a student), PBSPro did not process jobs correctly. The job switched into the state running but was not correctly started by PBS.

In all cases it was possible to simply manually restart the job after resolving of the failure. The experiment demonstrated that despite the fault tolerance support of Migol, it is not always possible to handle all possible failure mode. Nevertheless, Migol was able to detect and handle 96 % of the occurred failures correctly.

6 Related Work

Different frameworks for supporting the fault tolerance of Grid applications have been proposed, e. g. Condor-G [6], Condor/PGRADE [12], Cactus [13], HPC4U [8] and EasyGrid [2]. Unfortunately, these toolkits only address single aspects, e. g. job submission or checkpointing, particularly well. Many frameworks provide simple fault tolerance support, e. g. by supporting the monitoring and re-execution of jobs. But, these systems usually rely on very simplistic failure detection techniques, such as the polling of the job state. Migol provides as described an application-level failure detector, which is able to detect more complicated errors. Further, the architectures of these frameworks often introduce a single point of failure, e. g. by not considering a possible failure of the resource broker or the information service. It is very important to address fault tolerance within Grids from an overall system-wide perspective.

7 Conclusion and Future Work

Migol provides an integrated framework for addressing fault tolerance within distributed systems – from the application layer to the Grid infrastructure. The framework attempts to resolve failures transparently without human intervention. Migol is also able to detect and automatically react to infrastructure failures. Each service is designed with respect to fault tolerance considering the criticality of the service, the state managed by the service as well as the costs of the respective fault tolerance scheme. The architecture of Migol ensures that multiple services with the same capability exist. Only the highly critical core of Migol – the AIS – is actively replicated. Group communication is used to keep the state consistent across all replicas. Clients can use any replica for reading and updating their GSOs. For less critical services, i. e. the JBS, ARS, MS and CRS, service replicas exist, but no synchronization between those replicas is conducted. Further, these services are able to recover themselves. The Migol framework has been extensively tested with several real-life scientific applications on D-Grid and TeraGrid resources. The results of a long-term experiment demonstrated both the effectiveness and robustness of Migol.

This work addresses the inability of the MPI standard and the MPI implementations to precisely detect and handle failures. To avoid costly recoveries on Grid-level, it would be desirable to handle as much faults as possible on MPI-level. In order to enable applications and application frameworks to efficiently support fault tolerance on MPI-level, extensions to the MPI standard are necessary. The inclusion of fault tolerance features, such as failure detection, reconfiguration and checkpoint/recovery, is currently discussed as part of the MPI 3 standardization process. With the availability of enhanced fault tolerance support within MPI, the overall architecture of Migol will be revisited.

References

1. T. Banks. Web Services Resource Framework (WSRF) – Primer V1.2. <http://docs.oasis-open.org/wsrp/wsrp-primer-1.2-primer-cd-02.pdf>, 2006.
2. J. da Silva and V. Rebello. Low Cost Self-healing in MPI Applications. In *Proceedings of the 14th EuroPVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 144–152, Berlin, Heidelberg, September/October 2007. Springer.
3. M. Feller, I. Foster, and S. Martin. GT4 GRAM: A Functionality and Performance Study. In *Proceedings of the Teragrid 2007 Conference*, Madison, WI, USA, 2007.
4. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid. <http://www-unix.globus.org/toolkit/3.0/ogsa/docs/physiology.pdf>, 2002.
5. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. <http://www.globus.org/alliance/publications/papers/anatomy.pdf>, 2001.
6. J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, July 2002.
7. M. Gusowski, A. Luckow, B. Schnor, and M. Schütte. Experiences with a Resilient, MPI-based Master-Worker Application in a Failure-Prone Grid Environment. Technical report, University of Potsdam, Potsdam, 2008.
8. M. Hovestadt. Fault Tolerance Mechanisms for SLA-aware Resource Management. *11th International Conference on Parallel and Distributed Systems*, 2:458–462, 2005.
9. A. Iosup, M. Jan, O. Sonmez, and D. Epema. On the Dynamic Resource Availability in Grids. In *8th IEEE/ACM International Conference on Grid Computing (Grid 2007)*, April 2007.
10. J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
11. O. Khalili, J. He, C. Olschanowsky, A. Snavely, and H. Casanova. Measuring the Performance and Reliability of Production Computational Grids. In *GRID*, pages 293–300. IEEE, 2006.
12. J. Kovács and P. Kacsuk. A Migration Framework for Executing Parallel Programs in the Grid. In Marios D. Dikaiakos, editor, *European Across Grids Conference*, volume 3165 of *Lecture Notes in Computer Science*, pages 80–89. Springer, 2004.
13. G. Lanfermann. *Nomadic Migration – A Service Environment for Autonomic Computing on the Grid*. PhD thesis, University of Potsdam, Germany, 2003.
14. H. Li, D. Groep, L. Wolters, and J. Templon. Job Failure Analysis and its Implications in a Large-Scale Production Grid. *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 27, 2006.
15. A. Luckow, S. Jha, J. Kim, A. Merzky, and B. Schnor. Adaptive Replica-Exchange Simulations. *Royal Society Philosophical Transactions A*, to appear 2009.
16. A. Luckow and B. Schnor. Migol: A Fault-Tolerant Service Framework for MPI Applications in the Grid. *Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Application*, 24(2):142–152, 2008.
17. A. Luckow and B. Schnor. Service Replication in Grids: Ensuring Consistency in a Dynamic, Failure-Prone Environment. In *Proceedings of Fifth High-Performance Grid Computing Workshop in conjunction with IEEE International Parallel & Distributed Processing Symposium*, Miami, USA, 2008.
18. C. D. Ott, E. Schnetter, G. Allen, E. Seidel, J. Tao, and B. Zink. A Case Study For Petascale Applications in Astrophysics: Simulating Gamma-Ray Bursts. In *MG '08: Proceedings of the 15th ACM Mardi Gras conference*, pages 1–9, New York, NY, USA, 2008. ACM.
19. J. Schopf, L. Pearlman, N. Miller, C. Kesselman, I. Foster, M. D’Arcy, and A. Chervenak. Monitoring the Grid with the Globus Toolkit MDS4. In *Journal of Physics: Conference Series – Proceedings of SciDAC*, 2006.
20. B. Schroeder and G. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, 2006.
21. R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.

Efficient Simulation of Fork Programs on Multicore Machines

Jörg Keller¹, Christoph Kekler², and Bert Wesarg³

¹ FernUniversität in Hagen, Dept. of Math. and Computer Science, 58084 Hagen, Germany
`joerg.keller@fernuni-hagen.de`

² Linköpings Universitet, Dept. of Computer and Inf. Science, 58183 Linköping, Sweden
`chrke@ida.liu.se`

³ Techn. Univ. Dresden, Center for Inf. Services and High Performance Computing, 01062 Dresden, Germany
`bert.wesarg@tu-dresden.de`

Abstract. The SB-PRAM project resulted in the PRAM programming language Fork and in a complete tool chain from algorithms, libraries and a compiler to a prototype hardware and a sequential simulator of that prototype. With the prototype becoming defunct, the necessity arose to parallelize the simulator to be able to simulate larger Fork programs in a reasonable time on a multicore machine. We present an approach to increase the granularity of parallelism in the simulation and hence increase the speedup. The compiler forwards structural information of the Fork programming model via the code to the simulator. The simulator exploits this information at runtime to reduce synchronization overhead. We report on preliminary experimental results with a synthetic benchmark that indicate that our approach is advantageous.

Key words: PRAM simulation, granularity of parallelism, multicore programming

1 Introduction

The parallel random access machine (PRAM) is an important concept to study the inherent parallelism of problems and to devise parallel algorithms without consideration of or tweaking to architectural particulars of a target machine. While PRAM algorithms were considered unpractical for a long time, the SB-PRAM project provided the programming language Fork together with a compiler, a runtime system and the PAD library of basic algorithms on top of the SB-PRAM machine, a parallel architecture to efficiently execute Fork programs. In addition, the SB-PRAM simulator program provides support for Fork program development and debugging. See [1] for a detailed introduction into all of these issues.

Yet, as typically happens with prototypes, the SB-PRAM hardware became outdated quite fast and finally got disassembled in 2006. While the simulator remained, it is a sequential code and simulation of larger program instances would take days. As teaching of parallel algorithms and programming has become popular again because of multicore CPUs being rather the rule than the exception these days, and practical assignments should go beyond toy problems [2], we decided to provide a more powerful platform for Fork programming: a parallelized simulator program to emulate the SB-PRAM instruction set architecture on a contemporary shared memory parallel machine. While there are other PRAM languages such as *ll* [3], *e* [4] and XMTC [5], they either do not mention a simulator, or only provide a sequential simulator, so that we could hardly build on related work, except for parallel discrete event simulation in general [6].

In a previous attempt to parallelize this simulator [7] we explored the inherent parallelism between the PRAM processors but had to learn that this parallelism is very fine-grained, down to the level where the synchronization overhead is overall dominant. We were moderately successful in enlarging this granularity but could only achieve speedups on quite artificial program code.

In the present work we present a new approach to reduce the synchronization overhead by exploiting structural information available in the Fork programming model. Each Fork

program maintains a logical partitioning of the processors into groups. Initially all started processors run synchronously in a root group. Upon arrival of a **fork** statement or control flow statements such as **if** with a condition that depends on the processor ID (e.g. by using processor-local variables) the participating processors are split into subgroups, two in the latter case, more in the former case. At the end of the statement, the subgroups synchronize and join again to form the original group. The (sub)groups form a tree starting with the root group of all started processors. Only the leaf groups (which contain disjoint subsets of processors) of this tree are active. Fork only guarantees the synchronicity among the processors of each leaf group.

We have instrumented the Fork compiler to emit new instructions to communicate with the simulator in a one way fashion. These annotations signal the simulator when a group splits and when its subgroups will join again. With this information the simulator is capable to maintain the group tree by itself at runtime. The simulator uses this group tree to execute leaf node groups with low overhead. All PRAM processors in one leaf node group are simulated sequentially on one core round-robin instruction by instruction to maintain the synchronicity property of the Fork language, therefore avoiding any synchronization overhead of this fine-grained parallelism. Different leaf node groups can be executed in parallel without any synchronization because they are independent. Only in the case of group merges a synchronization is necessary. A side effect of this group monitoring approach is the avoidance of busy waiting when subgroups join again, so that already a 1-threaded version of our approach achieves a speedup over the previous simulator. With multi-threaded versions, we achieve a speedup over the previous simulator, and achieve a small speedup over the 1-threaded version for frequent situations. We explore the performance advantages with the help of a parameterizable synthetic benchmark.

The remainder of this article is organized as follows. In Sect. 2 we briefly summarize challenges in the simulation of PRAM programs that motivate our simulator design. In Sect. 3 we report on preliminary experimental results. Section 4 concludes.

2 Challenges in Parallel PRAM Simulation

The PRAM model assumes that all processors are executing their instructions synchronously, so this also is the semantics of the SB-PRAM instruction set architecture which is the target of the Fork compiler. In Fork all processors in the beginning execute the same instructions in an SPMD style like in MPI, i.e. they form a synchronous processor group. With the **fork**-statement, processors can be split into subgroups by evaluating an expression which is directly or indirectly dependent on the processor IDs. A similar situation with two subgroups appears when a group executes an **if**-statement with a condition somehow dependent on the processor IDs. Other situations where a group splits exist, such as a loop where the number of iterations is dependent on the processor IDs. The processors of each subgroup are synchronous, while the different subgroups are asynchronous. As a subgroup can be split again, at any time the groups form a group tree of which only the leaf node groups are currently active, and those leaves contain a partitioning of all PRAM processors.

When a group has split into subgroups, the subgroups unite again into the previous group after they have executed their block of instructions, such as **then** and **else** branches in the case of an **if**-statement. As the subgroups are asynchronous, they must pass a barrier to unite again. As the SB-PRAM was considered a single-user machine, a busy-waiting has been implemented.

In order to simulate the SB-PRAM sequentially, the processors are simulated round-robin, one instruction at a time. This guarantees the synchronous semantics but does not directly lead to a parallel simulation.

Note that there is also an asynchronous mode within Fork entered by the `farm`-statement. In this mode, all processors execute statements independently, i.e. they may become asynchronous. The user is responsible for taking appropriate action for coordination and avoiding race conditions, as is the case with other shared memory environments such as POSIX threads. Therefore, simulation on a multicore can be done by distributing the PRAM processors onto the cores and simulating larger blocks of instructions for each PRAM processor before switching to the next. As the asynchronous mode does not provide major challenges, we concentrate on the simulation of the synchronous mode.

In a previous attempt to parallelize the simulation for the synchronous mode [7], we have surveyed a number of possible approaches and finally implemented the time-warp optimistic simulation technique to perform a parallel simulation of all processors, taking access to shared memory as the events to be ordered by the time-warp simulation. However, the granularity of the available parallelism was so fine that only a moderate speedup could be achieved with a combination of techniques to coarsen that granularity, and on quite artificial benchmarks.

Therefore, we try a different approach here: The processors of each leaf node group are simulated sequentially as in the sequential simulator. The different leaf node groups can be simulated in parallel without overhead on different cores of our multicore platform as they are asynchronous. The PRAM shared memory is realized by allocation on the heap of the shared memory in the multicore platform. Access by PRAM processors of each group is serialized and hence the PRAM semantics is maintained. Access by different groups is not coordinated. The reasons are explained below. When a group is split into subgroups, the group tree representation in the simulator is extended, and the processors of the group are split onto the subgroups. As long as there are more threads than active leaf node groups, each group is assigned to a thread of its own. If there are more groups than threads, we use a runqueue of active groups served by the threads. When a group is joining again with another group, it is taken out of the runqueue until the other group has arrived as well. Thus, the busy waiting overhead is avoided. The threads are not terminated when groups join, but are re-used as we use a form of thread pooling. Also, there is no involuntary migration of groups between threads. One simulator thread executes a processor group until it either merges with its siblings or it splits itself into subgroups. Therefore the thread overhead is restricted to the split and merge of groups.

Unfortunately, the overhead for maintaining the group tree inside the simulator is not negligible in all cases. For example, if there are many leaf node groups that regularly merge again, then the threads regularly access the runqueue so that its lock has contention. Yet, in typical code with some structure the overhead is small as the results in Sect. 3 indicate.

In order to perform the operations above, the simulator must know when groups split and unite again. To achieve this, we enhanced the Fork compiler to instrument the generated code to provide this information to the simulator. The instrumentation comes in the form of additional instructions inserted by the compiler. We assigned a new instruction opcode to create the instrumentation instruction, because no existing instruction should be overloaded. This approach has the advantage that only the compiler backend had to be modified. The parallel simulator will not execute these instructions but act as described above.

The PRAM instruction set comprises read-modify-write instructions which are used as synchronization primitives between processor groups. These instructions were simulated with non-atomic instructions in the sequential simulator. In the multi-threaded simulator these instructions are implemented with atomic instructions provided by the host architecture. Using atomic memory access is not always required by the simulated code, but there is currently no information available whether the affected memory location will be accessed by multiple groups or not. This can lead to performance regressions in the simulator.

While two subgroups are asynchronous, they still both may access the same shared variable, such as in the following example code.

```
sh int a;
if (...private condition...) { // subgroup 1
    a = ...;
    ... = ... a ...;
} else { // subgroup 2
    ... = ... a ...;
}
```

Here subgroup 1 first writes into shared variable a , then a is read and the new value is used in the second assignment. Subgroup 2 reads a and uses it in the assignment. As Fork makes no assumption about the subgroups' progress relatively to each other, two cases can occur: either all processors in subgroup 2 read the old value of a , or all processors in subgroup 2 read the new value of a . Typically, we assume that an algorithm does not depend on which case actually occurs; we call such algorithms *robust*. Algorithms that are not robust may lead to race conditions depending on the simulation, so we do not consider them further.

Textbooks on PRAM algorithms such as [8,9,10,11] do not encounter this problem, as they do not treat hierarchical partitioning of processors in detail and consider all processors as one group executing one code synchronously, even when `if` statements with private conditions appear. Still, the algorithms presented there normally are robust.

3 Experimental Results

We use a synthetic benchmark to explore the benefits of our implementation. Although the code is rather simple, it covers a wide range of PRAM algorithms. The pseudo-code looks as follows:

```
times (4) {
    fork (#subgroups) {
        do_work(load +/- groupdev);
    }
    do_work(load * loadratio);
}
```

First, the p PRAM processors are split into an adjustable number of subgroups which all have the same size (assuming that `#subgroups` divides p .) Each subgroup then executes routine `do_work` with an adjustable `load` modified by variation `groupdev`. Then the subgroups synchronize and the root group (consisting of all p processors) executes routine `do_work` with the load scaled by parameter `loadratio`. The whole benchmark is repeated four times to even out runtime effects from other processes.

The parameter `load` serves to model the amount of useful work in contrast to the overhead of group splitting and joining. The parameter `groupdev` serves to model the runtime differences of different subgroups. The parameter `loadratio` serves to model the fraction of the simulation that is purely sequential because all PRAM processors form a single group.

All experiments simulate $p = 4096$ PRAM processors. We compare the sequential simulator with our approach using 1, 4 and 8 threads respectively, running on a quadcore processor (AMD Phenom II X4 920 processor, 4×2.8 GHz, 4×512 kB L2-cache, 6MB shared L3-cache.)

Fig. 1 depicts a situation with a very low workload of 16, no load deviation, and a strong sequential part with a load ratio of 50%, i.e. a situation quite uncomfortable for our approach. Still all versions achieve a speedup of about 1.4 for up to 256 subgroups, although we only see a small speedup of the multi-threaded versions over the single-threaded version for 2 and 4 subgroups, which however is a frequent case resulting from an if-statement (or two nested if-statements) with private condition. For large numbers of subgroups, the overhead in group tree administration strongly increases without providing appropriate benefit because the large number of subgroups leads to negative cache effects. The 8-threaded version is only slightly faster than the 4-threaded version in some situations, which indicates that already one thread per core is able to keep that core busy.

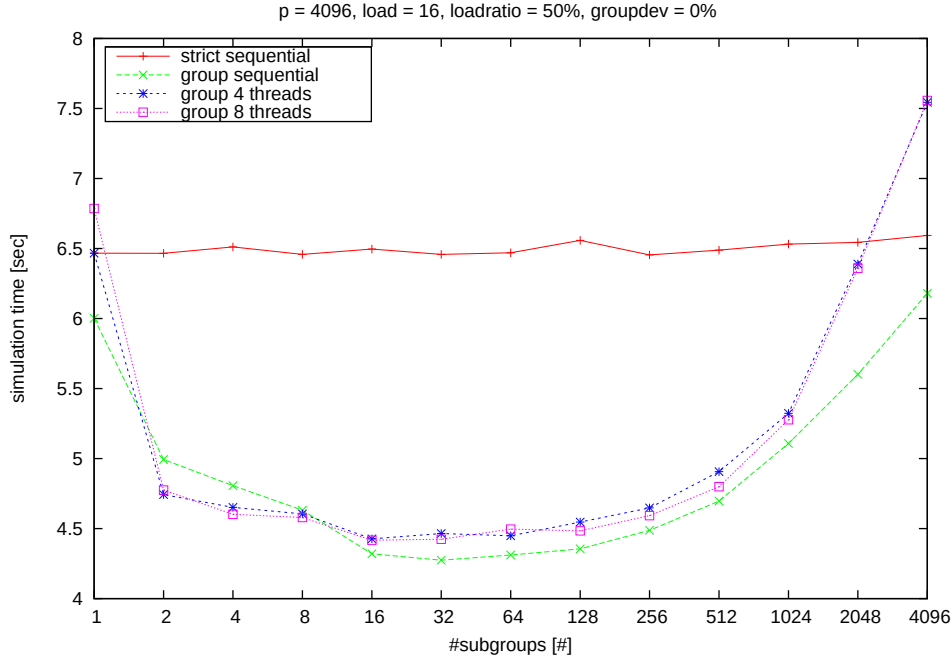


Fig. 1. Runtimes with low even workload and strong sequential part.

In Fig. 2 the load increases to 256. The situation is not really different from the previous one, which indicates that the group tree administration is not a bottleneck. In Fig. 3 we decrease the load ratio to 1% and see a sharp increase to a speedup of 2 over the sequential simulator, and a notable increase in the speedup of the multi-threaded versions over the single-threaded version for 2, 4 and 8 subgroups. In Fig. 4 we let the load deviate by up to 25%. We immediately notice that the runtime of the sequential simulator is increasing with the number of subgroups, because the load is not balanced anymore, and groups waiting for synchronization do a busy-wait. The speedup over the sequential version is larger than in any previous figure, because the groups waiting for a join with other groups are simply taken out of the run-queue until all groups participating in the join have arrived there. The speedup of the multi-threaded versions over the single threaded version for 2, 4 and 8 subgroups is still small but visible.

4 Conclusions

We have presented a new approach to accelerating the execution of Fork programs on multicore processors by exploiting the processor group structure in Fork. Our experimental

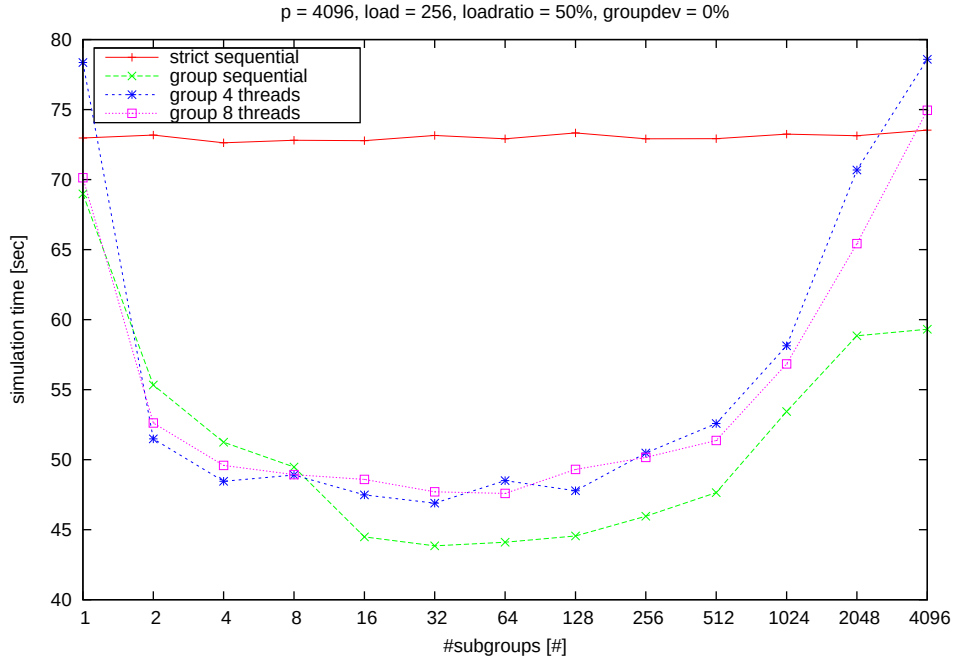


Fig. 2. Runtimes with workload increased compared to Fig. 1.

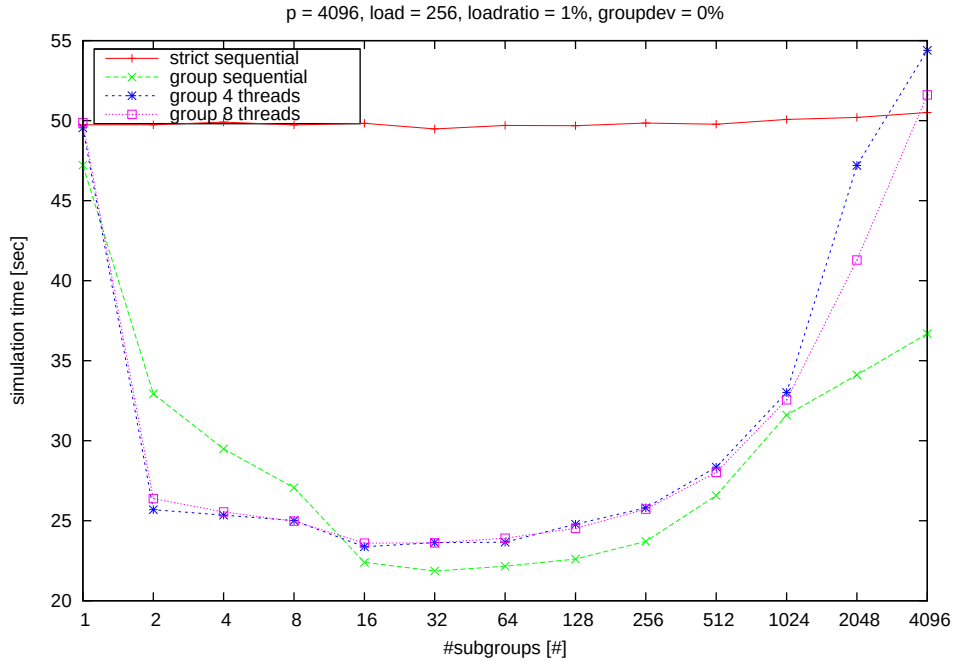


Fig. 3. Runtimes with sequential part decreased compared to Fig. 2.

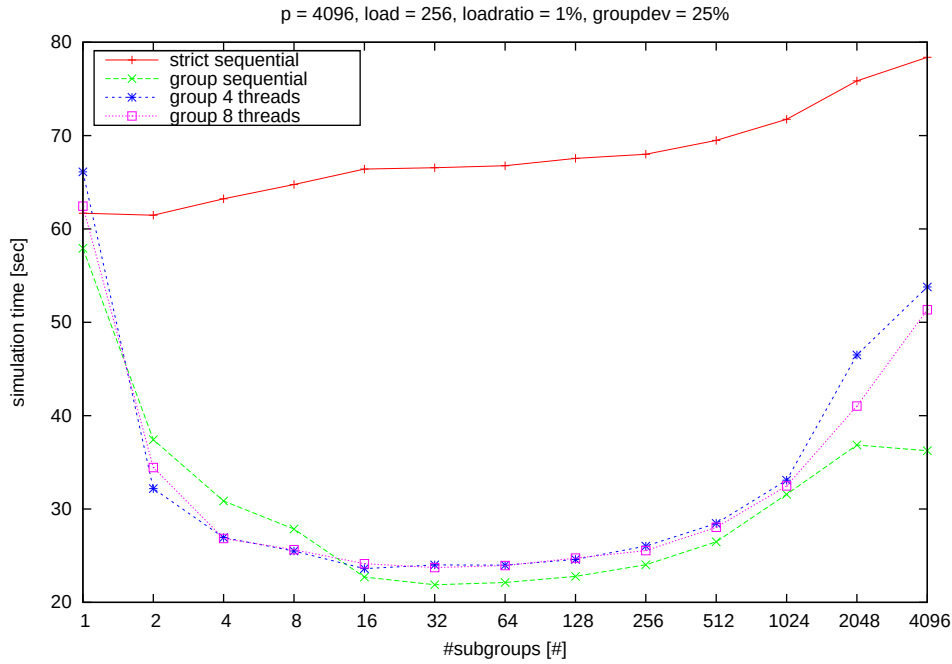


Fig. 4. Runtimes with load deviation increased compared to Fig. 3.

results indicate that already a single-threaded version shortens simulation time by a factor of 1.5 to 2. The multi-threaded versions achieve a small but noticeable speedup over the single-threaded version for the important case of a small number of subgroups.

For future work, another acceleration could be achieved by having the instruction stream for all processors of one group only decoded once and only execute the same instruction on different processors with different register contents as in a SIMD style.

It would also be nice to have a forecast on the simulation speedup to be expected when compiling the program. While this is possible in simple cases where the group tree structure is quite regular and not depending on input data, the general case would involve more or less a complete runtime analysis of the parallel program and thus seems out of reach.

References

1. Keller, J., Keßler, C.W., Träff, J.L.: Practical PRAM Programming. Wiley & Sons (2001)
2. Kessler, C.W.: A practical access to the theory of parallel algorithms. In: Proc. of ACM SIGCSE'04 Symposium on Computer Science Education. (2004)
3. León, C., Sande, F., Rodríguez, C., García, F.: A PRAM oriented language. In: Proc. EUROMICRO PDP'95 Workshop on Parallel and Distributed Processing. (1995) 182–191
4. Forsell, M.: e—a language for thread-level parallel programming on synchronous shared memory NOCs. WSEAS Transactions on Computers **3**(3) (2004) 807–812
5. Wen, X., Vishkin, U.: FPGA-based prototype of a PRAM-on-chip processor. In: CF '08: Proceedings of the 5th conference on Computing frontiers. (2008) 55–66
6. Fujimoto, R.M.: Parallel discrete event simulation. Communications of the ACM **33**(10) (1990) 30–53
7. Wesarg, B., Blaar, H., Keller, J., Kessler, C.: Emulating a PRAM on a parallel computer. In: Proc. 21st Workshop on Parallel Algorithms and Architectures (PARS 2007). (2007)
8. Akl, S.G.: The Design and Analysis of Parallel Algorithms. Prentice Hall, Englewood Cliffs, NJ (1989)
9. Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambridge University Press (1988)
10. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading, MA (1992)
11. Karp, R.M., Ramachandran, V.L.: A survey of parallel algorithms for shared-memory machines. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, Vol. A. Elsevier (1990) 869–941

Putting 'p' in RabbitCT - Fast CT Reconstruction Using a Standardized Benchmark

Hannes G. Hofmann, Benjamin Keck, Christopher Rohkohl, and
Joachim Hornegger

Pattern Recognition Lab, Friedrich-Alexander University Erlangen-Nuremberg
Martensstr. 3, 91058 Erlangen, Germany
{hannes.hofmann, benjamin.keck, christopher.rohkohl,
joachim.hornegger}@informatik.uni-erlangen.de
<http://www5.informatik.uni-erlangen.de/>

Abstract. Computational architectures and processors are an ever-changing field of research and development. Standardized and representable problem-dependent tests are required to find the optimal design of a running system. For the demanding problem of 3-D cone-beam tomographic reconstruction no such benchmarks are available to the public. We provide a standardized benchmark environment www.RABBITCT.com which enables the comparison of different hardware and system configurations. In this work we report the first results from a handful of different parallelization approaches. Among them software-based multi-threading, SIMD optimizations and state-of-the-art graphics cards using the CUDA programming environment.

Key words: Back-projection, Benchmark, CBCT, CT, GPU, Multi-core

1 Introduction

Clinical applications require fast 3-D reconstruction of tomographic data. Therefore, means to accelerate the reconstruction is a strong research topic. Currently the most wide-spread algorithms in clinical X-ray CT reconstruction belong to the class of filtered backprojection (FBP), e.g. the FDK method [1] for cone-beam data. Another class of reconstruction algorithms is based on iterative techniques which require multiple iterations of back- and forward projections. The latter can incorporate various corrections and provide superior image quality in certain cases like sparse or irregular data [2]. The fact that their complexity is a multiple of the FDK's might be a reason why this class of reconstruction algorithms is less commonly used in clinical systems.

The backprojection step is complex and bandwidth demanding – yet highly parallelizable. Hence, both classes of reconstruction algorithms can be accelerated well.

A manifold of specialized hardware and software techniques exist which can be applied to a multitude of reconstruction algorithms and acquisition scenarios. However, there exists no public means for comparing the speed and accuracy across different publications. We have realized the need for a standardized benchmark for reconstruction performance. Therefore, we have initiated RABBITCT [3, 4] which fills this gap (see Section 2.4). To test-drive this platform we have implemented a handful of modules with different parallelization approaches. Besides multi-threading using different libraries we present a hand-optimized SIMD implementation, a version combining both techniques and furthermore a GPU implementation using CUDA 2.1.

2 Methods and Materials

Over the last few years it became clear that the performance gain of new processors no longer comes from raising the clock rate. Instead, processor vendors have shifted paradigms and turn the increasing number of transistors into an increasing number of cores. Software developers have to write parallel code now to leverage all the available compute power. Multi-threading libraries (Sect. 2.1) help to utilize multiple cores. To also exploit the SIMD units on recent processors, the code has to be vectorized (Sect. 2.2). And finally, modern graphics cards are real work horses that can be employed by use of CUDA (Sect. 2.3) and the forthcoming OpenGL. This section is concluded by a description of our RABBITCT benchmark and the hardware used for our experiments.

2.1 Multi-Threading Libraries

Several APIs exist which support developers in multi-threaded programming. In this paper we have evaluated two of them, Intel's Threading Building Blocks (TBB) [5] and OpenMP [6]. We chose TBB because it will be supported on Larrabee [7], our next target platform, and OpenMP for its popularity and ease of use.

OpenMP is well established and supported by major hardware and software vendors. Code sections that should be executed in parallel are marked with a preprocessor directive. Thus compiler support is required which is provided with most modern compilers.

The open source project TBB was founded by its main contributor Intel. TBB is implemented as a C++ template library. It does not require compiler support and offers plenty of components for parallel programming.

2.2 Vectorization

Modern CPUs feature so-called vector processing units. Instead of performing one operation on a single element of data they can perform the same instruction on many elements concurrently, called SIMD (Single Instruction, Multiple Data). Current CPUs from AMD and Intel support SSE which can operate on four single precision floating point numbers at once. Future chips, e.g. Larrabee, will extend the SIMD width and be able to process up to 16 single precision elements simultaneously [7]. To utilize SIMD instructions, the programmer has to use intrinsics, which are translated into the corresponding assembler instructions by the compiler. Most modern compilers also try to automatically vectorize code – with varying success. The performance benefit of vectorization is greatest in parts of the program where many data elements are processed in the same way.

2.3 CUDA 2.1

In 2007, NVIDIA introduced their Common Unified Device Architecture (CUDA) [8], which was a fundamentally new programming approach at this time, making use of the unified shader design of the most current Graphics Processing Units (GPUs) from NVIDIA. The programming interface allows to implement an algorithm using standard C language and a few extensions. No knowledge about graphics programming using OpenGL, DirectX, and shading languages is necessary any more.

Since then CUDA evolved, up to CUDA 2.1 in the latest release, offering a stable system. It provides access to many hardware features, for the usage of high performance parallel computation. For example, NVIDIA’s QuadroFX 5600, which we used for our experiments, uses 128 stream processors in parallel and can additionally benefit from hardware accelerated features like texture interpolation. Furthermore, CUDA can be used on the latest and future graphics cards by NVIDIA.

Applying the easy to use C-like application programming interface of CUDA, the following two different parts in each CUDA implementation have to be distinguished: A host part, which executes in a CPU thread, and a device part (kernel). The latter is invoked by the controlling CPU thread, but runs in parallel on the GPU device. Due to the fact that the kernel can only operate on the graphics card’s memory, the API provides functions for memory management from within the CPU thread.

2.4 RabbitCT

To evaluate different approaches to exploiting parallelism, we implemented several modules for the benchmark suite RABBITCT using different optimization techniques. RABBITCT provides a standardized C-arm CT dataset, problem statements and framework to benchmark the performance of the backprojection step of the FDK algorithm. The dataset consists of $N = 496$ pre-processed projection images $\mathbf{I}_n \in \mathbb{R}^{S_x \times S_y}$, $n = 1 \dots N$ from a C-arm system (Siemens AG, Artis Zee) acquired on a 200° circular short-scan trajectory. The size of a projection image is $S_x = 1248$ pixels in width and $S_y = 960$ pixels in height at an isotropic resolution of $0.32 \frac{\text{mm}}{\text{pixel}}$.

For each projection image a pre-calibrated projection matrix $\mathbf{A}_n \in \mathbb{R}^{3 \times 4}$ is available. It encodes the perspective projection, in homogeneous coordinates, of a 3-D object point onto the 2-D projection image [9, 10].

The task for RABBITCT modules is the reconstruction of an isocentric cubic volume of 256^3 mm^3 . We defined four different problem sizes with different computational costs. The side lengths of the cubic reconstruction are $L \in \{128, 256, 512, 1024\}$ voxels respectively at an isotropic voxel size of $R_L = \frac{256}{L} \text{ mm}$.

2.5 Test Hardware

CPU	Sockets × Cores	Clock Speed	L2 Cache	L3 Cache	RAM
Core2Duo T9300	2	2.50GHz	6MB	–	4GB
Core2Extreme QX9650	4	3.00GHz	2 × 6MB	–	4GB
Xeon X7460	4 × 6	2.67GHz	12 × 3MB	4 × 16MB	32GB
Nehalem-EP	2 × 4	2.67GHz	8 × 256KB	2 × 8MB	12GB

Table 1. Specifications of our test systems. Missing in this list is the graphics card, see text for details.

The most important specifications of our test systems are listed in Table 1. Note that the machines with Xeon X7460 (Dunnington) and Nehalem processors were pre-production systems. We expect production hardware to deliver similar performance levels. Missing in this list is the graphics card used for the CUDA benchmarks. We used a NVIDIA QuadroFX 5600 installed in a Fujitsu Siemens Celsius R650 with an Intel Xeon E5410 CPU and 16 GB of main memory.

Nehalem's two major novelties are the QuickPath Interconnect (QPI) which replaces the legacy Front Side Bus (FSB) and the integrated memory controller with support for DDR3 SDRAM. These two result in a substantial increase in main memory bandwidth.

For tests on the Core2Duo (C2D) and Core2Extreme (C2E) systems, the MSVC9 compiler was used. The Dunnington and Nehalem systems were benchmarked using both gcc in version 4.3.3 and the Intel compiler in version 11.0.074.

3 Implementation

Algorithm 1 describes the core part of the FDK algorithm. It is executed for every projection image I_n . The three inner loops (i, j, k) traverse the volume in x , y and z direction and update each voxel exactly once.

Algorithm 1: The backprojection kernel for the n -th projection image.

```

input :  $I_n, A_n, L, O_L, R_L, f_L$ 
output: updated reconstruction  $f_L$ 

for  $k = 0$  to  $L - 1$  do
     $z = O_L + kR_L$ ;
    for  $j = 0$  to  $L - 1$  do
         $y = O_L + jR_L$ ;
        for  $i = 0$  to  $L - 1$  do
             $x = O_L + iR_L$ ;

            // Update the volume
             $f_L(i, j, k) + = \frac{1}{w_n(x, y, z)^2} \cdot \hat{p}_n(u_n(x, y, z), v_n(x, y, z));$ 
        end
    end
end

```

3.1 Multi-Threading

Within one iteration of the projections-loop the FDK algorithm has no competing write accesses. Therefore, any of the x , y , z loops can be split into disjoint ranges and executed in parallel.

To reduce the overhead of thread management we decided to parallelize only the outermost (z) loop. Of course, the actual number of chunks into which the loop is divided is usually equal to N_P , the number of compute cores. This method will scale up to N_z cores, where N_z is the number of voxels in z -direction (128, 256, 512 or 1024). As N_P is typically small compared to N_z , we can be sure that each task is big enough to compensate the start-up overhead. This strategy was used for both the OpenMP and TBB implementation.

In our OpenMP implementation we defined a parallel work-sharing region (`#pragma parallel for`) around the for-loop which iterates in z -direction.

The TBB implementation uses `parallel_for` as well. But due to the design of TBB the kernel had to be wrapped in a C++ class which stores the state of the current task. The TBB scheduler creates one instance of this class per thread and initializes it with the correct boundaries. Then its `operator()` method is called by the scheduler.

3.2 Vectorization

Vectorization is most efficient in code parts where a lot of computation is done. The y and z loops do only increment the coordinates but the actual backprojection, including a bilinear interpolation, happens within the x -loop. It iterates over all voxels in one column (fixed y and z coordinate) and performs the same computation on each of them. This makes it just natural to use SIMD here.

While multi-threading was supported by the libraries, vectorization was done manually. For many parts of the code it was straightforward. However, vectorization is particularly complicated for code sections that contain conditional branches. There is a check if the current voxel is in the Field-of-View (FOV) of the current projection before the projection value is read. If it lies outside, the voxel update is skipped and processing continues with the next voxel. Since SIMD units can not branch independently for individual vector elements all voxels have to be updated. To avoid invalid memory accesses the projected coordinates of a voxel are set to $(0, 0)$ if they lie outside of the projection. After fetching the projection values, the outliers are set to 0. Both measures can be implemented efficiently using multiplicative masks.

Another problem is caused by the geometry. Rays through adjacent voxels do not necessarily end up in neighboring pixels in the projection image. The result are non-linear memory accesses when loading the projection values for a vector of voxels. Therefore, all pixel values have to be fetched in a scalar manner and inserted into vectors which are then used to compute the bilinear interpolation.

3.3 GPU / CUDA 2.1

In our CUDA 2.1 implementation we invoke our backprojection kernel on the graphics device. In CUDA the parallelization is originated due to the partition of the problem into a grid of blocks, while each block consists of a limited number of threads. Each thread of the kernel computes the backprojection of a certain column of the volume in y -direction. This means that the volume is parallel processed for each x - z -coordinate.

This allows to save six multiply-add operations by incrementing the homogeneous coordinates with the appropriate column of the projection matrix for neighboring voxels in y -direction. This approach also reduces the register usage down to 9 registers for a backprojection kernel. The detailed description of the CUDA implementation can be found in [11].

4 Results

We ran benchmarks for volumes of 128^3 , 256^3 and 512^3 voxels. Most emphasis is put on the results of the largest problem as they are relevant for clinical use. Moreover, the start-up overhead has more impact on the overall runtime on smaller volume sizes. Therefore, if not said explicitly we refer to the 512^3 volume.

RABBITCT measures the plain backprojection runtime without reading from and storing to hard disk. We anticipate a situation where we either have sufficient hard disk speed (e.g. using a RAID system) or get the projections on the fly over a fast network link. To accomplish this, we tried to load all projections to main memory before backprojecting all of them. However, due to memory limitations on some systems, we had to split the projections into four sets and process the

sets sequentially. For the CUDA implementation, the final transfer of the reconstructed volume from the GPU to the host’s main memory is not included in the measurements because typically the volume would be rendered after reconstruction and could therefore stay on the GPU. The times given in this section are the mean backprojection time for one projection. Runtimes are actual measurements, speed-ups comparing different systems are normalized by the CPUs’ clock speed.

Volume Size	Core2Extreme		Nehalem	
	Runtime	Factor	Runtime	Factor
128^3	7.20	$1.00\times$	3.25	$1.00\times$
256^3	49.76	$6.91\times$	22.16	$6.81\times$
512^3	400.38	$55.59\times$	170.74	$52.46\times$

Table 2. Mean backprojection time in *ms* on the Core2Extreme and Nehalem systems for different volume sizes. Speed-ups are relative to the 128^3 volume. Implementation: TBB+SIMD.

Table 2 shows how the runtime changes with different volume sizes. Theoretically the complexity is increased by a factor of 8 when the length of the reconstructed volume is doubled. However, when increasing the volume size from 128^3 to 256^3 voxels, the factor is significantly less than 8. This indicates that the start-up overhead is a substantial part of the total computation time in the 128^3 case. In contrast, the factor is almost 8 when doubling cube length again to 512^3 . The problem size of 256^3 is sufficient to hide the overhead.

Implementation	Core2Duo	Core2Extreme	Nehalem	Dunnington
single-threaded	6322.27	5389.05	5041.82	5064.11
SIMD	1789.47	1505.71	1711.64	1694.94
OpenMP	3126.36	1256.71	456.41	229.52
TBB	2683.81	1119.96	447.50	222.32
TBB+SIMD	990.99	400.38	170.74	116.47

Table 3. Mean backprojection time in *ms* on selected systems for different optimizations. The volume size was 512^3 .

Volume Size	CUDA	Dunnington
128^3	4.00	2.70
256^3	10.14	13.55
512^3	34.50	116.47
1024^3	—	931.83

Table 4. Mean backprojection time in *ms* on the GPU and on the fastest CPUs. A volume of size 1024^3 was not too large to store on the used graphics card.

Table 3 provides an overview over the measurements of all CPU-based implementations with a 512^3 volume on different systems. On the Nehalem and Dunnington systems the Intel compiler was used, on the Core2Duo and Core2Extreme MSVC. The results of the CUDA implementation can be found in Table 4.

Table 5 shows the achieved speed-ups for different optimizations. The speed-up with OpenMP was $18.8\times$ on the Dunnington system and $10.2\times$ on the 8-core Nehalem using gcc. The Intel compiler performed even better achieving $22.1\times$

Implementation	Core2Duo Speed-up	Core2Extreme Speed-up	Nehalem Speed-up	Dunnington Speed-up
single-threaded	1.00×	1.00×	1.00×	1.00×
SIMD	3.53×	3.58×	2.95×	2.99×
OpenMP	2.02×	4.29×	11.05×	22.06×
TBB	2.36×	4.81×	11.27×	22.78×
TBB+SIMD	6.38×	13.46×	29.53×	43.48×

Table 5. Shows impact of different optimizations on the same CPU, for two different CPUs. Speed-up is relative to single-threaded implementation on same CPU.

Implementation	Compiler	Dunnington Runtime [ms]	Nehalem Runtime [ms]
OpenMP	gcc	269.23	494.95
OpenMP	icc	229.52	456.41
TBB	gcc	224.23	447.12
TBB	icc	222.32	447.50

Table 6. Comparing gcc and the Intel compiler on two systems. The volume size was 512^3 . Note the improvement of the OpenMP implementation when using icc.

and $11.1\times$ respectively. As mentioned in Section 3.1 TBB required more effort to implement. But the reward was better performance throughout all our experiments. The speed-up with TBB was $22.6\times$ and $11.3\times$ respectively, also using gcc, and $22.8\times$, $11.3\times$ using the Intel compiler. When using MSVC or gcc, the advantage of TBB was between 10 and 20%. Even with the Intel compiler’s optimized OpenMP implementation, TBB is still 2 – 3% ahead (cf. Tables 5 and 6).

When evaluating the performance gain from vectorization (cmp. Table 5), one thing is noticeable. For the Core2* systems, the speed-up is about $3.5\times$, but for the Nehalem and Dunnington it’s only around $3\times$ (cf. Table 5). Our explanation is that the Intel compiler does a pretty good job in auto-vectorizing and code optimization, resulting in a higher baseline. The next thing is that the speed-up from the TBB implementation to TBB+SIMD is noticeably lower than from baseline to SIMD. While the Core2* and Nehalem systems achieve only 2.6 to $2.8\times$, the speed-up is even less on the Dunnington system ($1.9\times$). This is due to the bandwidth-bound nature of our algorithm. The higher number of threads consumes the total available memory bandwidth before the peak computation rate is reached. A similar result could be shown in previous experiments on GPUs (not yet published).

The results in Table 7 come up to the expectations of the Nehalem platform. Having only twice as much cores as the Core2Extreme the speed-up was between 2.63 for TBB+SIMD and $3.09\times$ for OpenMP. One could argue that a good part of this effect is due to the Intel compiler. But with gcc the speed-up is 2.58 to $2.85\times$ – still significantly higher than $2\times$. This is explained by the bandwidth-boundedness of the algorithm. The higher main memory bandwidth results in super-linear speed-ups.

As one can see in Table 4 the CUDA implementation suffers from its start-up overhead and is considerably slower than the Dunnington system for the volume size of 128^3 voxels. But it is already faster by a factor of $1.34\times$ for 256^3 volumes and gains an even bigger advantage on 512^3 volumes where it outperforms the fastest CPU-based system by a factor of $3.38\times$.

Implementation	Core2Duo Speed-up	Core2Extreme Speed-up	Nehalem Speed-up	Dunnington Speed-up
single-threaded	1.02×	1.00×	1.20×	1.20×
SIMD	1.01×	1.00×	0.99×	1.00×
OpenMP	0.48×	1.00×	3.09×	6.15×
TBB	0.50×	1.00×	2.81×	5.66×
TBB+SIMD	0.48×	1.00×	2.63×	3.86×

Table 7. Comparing performance of different CPUs. Speed-up is relative to C2E and was normalized by clock rate.

5 Discussion

Using current libraries, multi-threading can be implemented easily if the problem has such a promotive nature as the backprojection algorithm. The bottom line for both multi-threading libraries investigated is that they scale nicely with the number of cores (when normalized by the clock speed).

Vectorization on the other hand is more difficult to implement. It is a great way to get more performance out of existing hardware. But compared to the speed-ups from multi-threading, the effort of vectorization looks not worthwhile. This will change in the future, when chips with wider SIMD units become available. Our experiences have shown that the step from 4-wide SIMD to 16-wide requires only minor modifications of the source code.

Different CPUs behave numerically equivalent. The results using SSE instructions differ only negligible from scalar computations. This is shown by the worst mean squared error (MSE) of all CPU implementations, which was only 0.001 (range: 0...4095). The CUDA implementation deviated from the reference implementation by about 8.1 HU². This error is still acceptable for clinical purposes. Note that since we reconstructed real acquired data we do not even know the true values. Thus we can not say if the GPU result is better or worse. But this example shows that the accuracy of different hardware platforms is an issue that should be further investigated. Accuracy measurements using simulated phantoms would be required to judge the image quality.

One reason that graphics cards are so fast is that they have fast on-board memory which offers high bandwidth. Its size is typically in the order of 1 or 2 GB. Volumes that do not fit in this amount of memory have to be split into several parts which are reconstructed independently. This is not only a cumbersome task but also involves transfers of huge amounts of data from and to the computer's main memory. If these transfers cannot be hidden using double buffering techniques they will decrease the reconstruction speed.

6 Conclusion and Outlook

In this paper we have compared five different approaches to exploiting the inherent parallelism of the backprojection algorithm. Each of them shows different potential for accelerating reconstruction algorithms. Their common denominator is that they are limited by the main memory bandwidth.

There exist further optimization methods which we did not investigate in this work. With loop unrolling and software pipelining the compiler and the developer

can try to hide latencies of main memory accesses. Moreover, cache-aware implementations can take advantage of the large L2 (and L3) caches of the most recent CPUs. They can keep data close to the compute unit for a longer time and re-use it more often before it is written back to main memory.

The RABBITCT benchmark framework allows rapid development and assessment of different backprojection implementations. It is a suitable platform for prototyping and performance comparison of backprojection algorithms. We used it to evaluate maximum system performance only but more detailed measurements on scalability with only a small number of threads could be performed as well.

A variety of different hardware architectures exists to date. And the diversity will further increase. Future hardware generations (Intel's Larrabee, AMD's Fusion) will bring more heterogeneity. This makes it harder for programmers to exhaust their full power.

Developers are therefore supported by different parties. Modern compilers provide auto-vectorization options which are improved with every release. Software vendors develop tools that analyze code for parallelism and suggest threading strategies. Others provide new tools for debugging of multi-threaded programs. Besides the two libraries used in this work, more frameworks exist to ease multi-threaded programming. One example, Cilk++ [12], shows similarities to TBB. A special compiler transforms annotated sequential code into a parallel program. Sequoia [13] on the other hand strives to enable the programmer to create programs which are aware of the memory hierarchy and make the most of it. However it seems like this project is not under development any more.

Different libraries and novel platforms will be subject of our upcoming research. The recent results will be available online at RABBITCT [4]

Acknowledgements. This work was supported by the Regional Computing Center Erlangen (RRZE) and Intel Corporation who kindly provided the authors access to the pre-production systems.

References

1. Feldkamp, L., Davis, L., Kress, J.: Practical Cone-Beam Algorithm. *Journal of the Optical Society of America* **A1**(6) (1984) 612–619
2. Kunze, H., Härer, W., Stierstorfer, K.: Iterative extended field of view reconstruction. In Hsieh, J., Flynn, M.J., eds.: *Medical Imaging 2007: Physics of Medical Imaging*. Volume 6510 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, San Diego (2007) 65105X
3. Rohkohl, C., Keck, B., Hofmann, H.G., Hornegger, J.: Technical note: Rabbitct—an open platform for benchmarking 3d cone-beam reconstruction algorithms. *Medical Physics* **36**(9) (2009) 3940–3944
4. Chair of Pattern Recognition, Friedrich-Alexander University Erlangen: RabbitCT, Open Platform for Worldwide Comparison in Backprojection Performance (2009)
5. Intel Corp.: *Threading Building Blocks* (2009)
6. OpenMP: The OpenMP API specification for parallel programming. Website (2009) Available online at <http://www.openmp.org/>.
7. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. In: *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, Los Angeles, ACM (2008) 1–15
8. NVIDIA Corp.: *NVIDIA CUDA Compute Unified Device Architecture Programming Guide* (2007)
9. Faugeras, O.: *Three-Dimensional Computer Vision (Artificial Intelligence)*. The MIT Press Cambridge (1993)
10. Wiesent, K., Barth, K., Navab, N., Durlak, P., Brunner, T., Schuetz, O., Seissler, W.: Enhanced 3-D-reconstruction algorithm for C-arm systems suitable for interventional procedures. *IEEE Transactions on Medical Imaging* **19**(5) (2000) 391–403
11. Scherl, H., Keck, B., Kowarschik, M., Hornegger, J.: Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA). In Frey, E.C., ed.: *Nuclear Science Symposium, Medical Imaging Conference 2007*. Volume 6., Honolulu (2007) 4464–4466

12. Frigo, M.: Multithreaded Programming in Cilk. In: Proceedings of the 2007 international workshop on Parallel symbolic computation, New York, NY, USA, ACM (2007) 13–14
13. Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming the memory hierarchy. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. (2006)

TBCA

Eine Thread-basierte Cluster-Architektur

Martin Meierhöfer, Bernd Klauer

Technische Informatik, Fakultät für Elektrotechnik
Helmut-Schmidt-Universität Hamburg / Universität der Bundeswehr Hamburg
Holstenhofweg 85, 22043 Hamburg, Germany

Zusammenfassung Durch den Einsatz von Multicore-Prozessoren werden aus Arbeitsplatzrechnern Parallelrechner. Um die Leistungsfähigkeit dieser Systeme ausnutzen zu können, ist der Einsatz paralleler Programmier Techniken notwendig. Threads sind hierfür besonders geeignet, da der entstehende Verwaltungsaufwand gering ist. Aus mehreren Multicore-Systemen lässt sich außerdem auch schnell und kostengünstig ein Cluster zusammenstellen. Der Anspruch an den Programmierer wächst damit jedoch enorm. Mit Threads alleine lässt sich solch ein Cluster nicht mehr programmieren. Mehrere Techniken sind notwendig, die aufeinander abgestimmt werden müssen. Oftmals lässt sich eine Anwendung nur für einen bestimmten Cluster optimieren.

Die TBCA stellt eine Mittelschicht dar, die die Verteilung von Threads in einem Cluster ermöglicht. Neben einer automatischen Verteilung kann der Programmierer gezielt auf die Verteilung Einfluss nehmen und somit die Anwendung optimieren. In dieser Arbeit wird die TBCA und der derzeitige Stand der Implementierung unter Linux vorgestellt.

Key words: pthreads, cluster, multicore

1 Einleitung

Parallele Programmierung ist heutzutage im Bereich der Hochleistungsrechner unentbehrlich. Aber auch bei günstigen Servern und bei Arbeitsplatzrechnern gewinnt die Parallelisierung immer mehr an Bedeutung. Nach den Dualcore-Prozessoren sind nun Multicore-Systeme mit drei und vier Prozessoren erhältlich. Mit diesen lassen sich Standardserver derzeit mit bis zu 16 Rechenkernen ausstatten. Diese Systeme können kostengünstig mit Netzwerken bis zu 1 GBit/s vernetzt werden. Damit lassen sich relativ preiswert Cluster mit einer hohen Rechenleistung zusammenstellen. Die Taktraten der einzelnen Prozessoren stagnieren derzeit. Daher kann die Rechenleistung eines solchen Rechnerverbundes nur durch die parallele Implementierung von Anwendungen genutzt werden.

Für den Programmierer reicht neben der sequentiellen Programmierung eine einfache Parallelisierung nicht mehr aus. Es sind verschiedene Techniken wie beispielsweise MPI [1], OpenMP [2], POSIX-Threads [3], PVM [4], SDVM [5] oder Mosix [6] notwendig. POSIX-Threads lassen sich beispielsweise nur auf einem System einsetzen und MPI nutzt nur einen Rechenkern pro Clusterknoten. Bei dem Einsatz mehrerer dieser Techniken ergibt sich dann zumeist eine Applikation, die lediglich für einen bestimmten Cluster optimiert ist.

Die Thread-Based-Cluster-Architecture (TBCA) verteilt POSIX-Threads in einem Cluster. Damit benötigt der Programmierer zunächst nur eine Technik - POSIX-Threads -, um eine Applikation auf einem Cluster ablaufen zu lassen. Die Verteilung auf die zur Verfügung stehenden Rechenkerne geschieht automatisch. Als Ergänzung kann der Programmierer mit Hilfe von Metainformation Einfluss auf die Verteilung nehmen. Ziel ist es, eine Middleware zur Verfügung zu stellen, die eine einfache Programmierung erlaubt. Zur Steigerung der Performance sollen die automatischen Mechanismen jedoch gezielt beeinflusst werden können.

2 Allgemeine Architektur

Ähnlich wie bei der OpenCL-Spezifikation der Khronos Group [7], besteht die TBCA aus einem Master und mehreren Clients (bei OpenCL *Host* und *OpenCL Devices* genannt). Sowohl auf dem Master als auch auf den Clients laufen Dienste. Diese sind für die Verteilung der Threads zuständig. Allerdings wird keine eigene Programmiersprache wie *OpenCL C* benötigt. Der Quelltext einer Thread-basierten Applikation wird durch einen Präprozessor vorverarbeitet. Der entstehende Quelltext kann von einem herkömmlichen C-Compiler übersetzt werden.

Der TBCA-Präprozessor modifiziert den Quelltext so, dass das Programm auf zwei Arten ausgeführt werden kann. Zum einen wie bisher als Hauptprogramm einer Thread-basierten Applikation. Andererseits besteht die Möglichkeit, dass lediglich ein Thread des Programms ausgeführt wird. Dazu wird eine neue *main*-Funktion (im folgende *tbca-main*) eingesetzt. Die alte Funktion *main* (im folgenden *org-main*) bleibt erhalten, wird jedoch nicht mehr zu Programmstart ausgeführt. Für jede Funktion die als Thread ausgeführt werden soll, wird eine weitere Funktion (im folgenden *tbca-thread*) eingefügt.

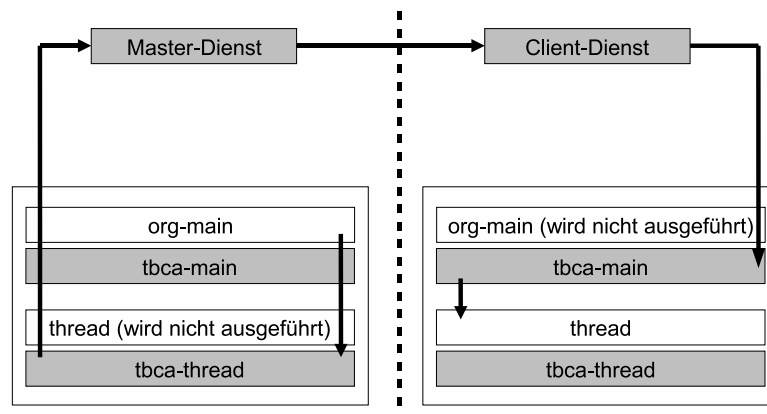


Abbildung 1. Starten eines dislozierten Threads

Wird das Programm nun gestartet, prüft *tbca-main*, ob sich das Programm auf dem Master oder auf einem Client befindet. Auf dem Master wird *org-main* ausgeführt. Soll im Laufe des Programms ein Thread gestartet werden, wird an dessen Stelle der entsprechende *tbca-thread* gestartet. Dieser meldet am Dienst des Masters den zu startenden Thread an (siehe Abbildung 1). Der Master-Dienst entscheidet, auf welchem Rechenkern der Thread ausgeführt wird. Der entsprechende Client-Dienst startet das Programm auf dem Client und bindet das Programm - und damit den auszuführenden Thread - mit der Funktion *sched_setaffinity* an den entsprechenden Rechenkern. Auf dem Client wird nun ebenfalls *tbca-main* ausgeführt. Diese startet nun den Thread.

Die Verteilung der einzelnen Threads übernimmt dabei ausschließlich der Master-Dienst. Er kennt alle Clusterknoten mit den zur Verfügung gestellten Rechenkernen und deren Auslastung. Der Client-Dienst hat die Aufgabe, den Clusterknoten am Master mit der zur

Verfügung stehenden Anzahl an Rechenkernen bekannt zu machen und nach Aufforderung durch den Master einen dislozierten Thread zu starten. Bei Bedarf kann die Kommunikation zwischen einem Programm und einem ausgelagerten Thread über den Master- und Client-Dienst laufen.

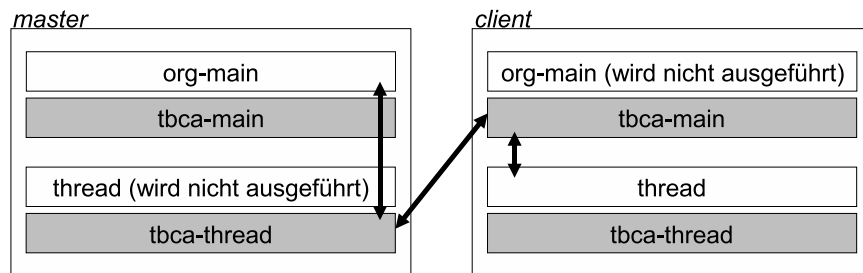


Abbildung 2. Kommunikation zwischen Hauptprogramm und Thread

Die Kommunikation zwischen dem Thread und dem Hauptprogramm erfolgt über *tbca-thread* auf dem Master und *tbca-main* auf dem Client (siehe Abbildung 2). Über diese Verbindung werden beispielsweise Variablen abgeglichen, Mutexe und Semaphore gesetzt. Die Clients müssen hierbei auf die Umsetzung auf dem Master warten. Hierbei zeigt sich bereits, dass die verteilten Threads möglichst unabhängig und mit langer Laufzeit implementiert werden sollten.

Der Programmierer kann dem Quelltext Metainformationen in Form von Kommentarzeilen hinzufügen. Diese werden vom TBCA-Präprozessor ausgewertet. Auf diese Weise kann der Entwickler beispielsweise festlegen, dass ein Thread in jedem Fall auf dem Master ausgeführt werden soll. So kann er in seine Anwendung sowohl lang und möglichst unabhängig laufende Threads zur Verteilung auf andere Clusterknoten als auch kurze und eng angebundene Threads zum Ablauf auf dem Master implementieren und entsprechende Metainformationen der TBCA zur Verfügung stellen.

In einer späteren Entwicklungsstufe sollen über die Dienste auch Informationen über die einzelnen Knoten gesammelt werden. Dann wäre auch denkbar, dass der Programmierer die Art des zu verwendenden Rechenkerns für einen Thread festlegen kann (z.B. CPU mit Stärken in der Berechnung von Gleitkomma-Operationen).

3 Speichermanagement

Eine große Herausforderung bei verteilten Applikation ist die Verwaltung des gemeinsamen Speichers. Threads auf einem einzelnen System haben dadurch große Vorteile, dass sie sich einen gemeinsamen Speicherbereich teilen. Dies ermöglicht, dass ein Thread schnell erzeugt werden kann. Soll ein Thread nun auf einen anderen Knoten ausgelagert werden, so muß dorthin nicht nur der gesamte Kontext des Prozesses übertragen, sondern auch aktuell gehalten werden.

Für die Bearbeitung des Quelltextes durch den TBCA-Präprozessor können die verwendeten Variablen in drei Kategorien eingeteilt werden. Lokale Variablen werden nur durch den Thread selbst verwendet. Hier ist keine Synchronisation notwendig. Globale Variablen müssen synchronisiert werden. Der Zugriff auf globale Variablen sollte bei der Verwendung von Threads allerdings durch Mutexe abgestimmt werden. Somit kann das Sperren und das Freigeben eines Mutex als Signal zur Synchronisation verwendet werden. Zu anderen Zeitpunkten ist eine Synchronisation dann nicht mehr notwendig. Die TBCA schreibt deshalb für den Zugriff auf globale Variablen die Nutzung von Mutexen vor.

Zeiger stellen die dritte Kategorie dar. Hier ist eine automatische Synchronisation schwierig. Der Zeitpunkt der Synchronisation kann wie bei den globalen Variablen an den Mutex gebunden werden. Allerdings ist bei einem Zugriff über einen Zeiger die Größe des Speicherbereichs nicht immer erkennbar (z.B. Felder mit dynamischer Größe). Aus diesem Grund muß der Programmierer einen solchen Zugriff als Metainformation ankündigen.

Hierfür werden vom TBCA-Präprozessor Metainformationen in Kommentarzeilen ausgewertet. Drei verschiedene Arten solcher Informationen sind derzeit möglich.

```
//tbca_lock varname bytes
//tbca_unlock varname
//tbca_priority number
```

Mit `tbca_lock` werden ab der Adresse von `varname` die vorgegebene Anzahl an Bytes im Speicher gesperrt. Diese Sperre wird mit Hilfe von `pthread_mutex_lock()` auf dem Master umgesetzt. Mit `tbca_unlock` wird eine solche Sperre aufgehoben. Das Konzept, Metainformationen in Kommentarzeilen zu übergeben, wird auch für das Scheduling der Threads verwendet. Mit Hilfe von `tbca_priority` kann (wie oben beschrieben) Einfluß auf die Auslagerung von Threads genommen werden. Dieses Konzept ermöglicht, dass der Quelltext auf einem System ohne TBCA übersetzt und ausgeführt werden kann.

4 Sicherheit

Das Hauptziel der TBCA ist es, eine Applikation performant ablaufen zu lassen. Daher findet die Kommunikation zwischen dem Hauptprogramm und einem ausgelagerten Thread normalerweise direkt statt. Die TBCA kann auch so konfiguriert werden, dass die Kommunikation über den Master- und Client-Dienst läuft und zusätzlich mit AES verschlüsselt wird. Dies ermöglicht, dass durch den Einsatz von Firewalls ein Großteil der Ports gesperrt wird und damit der einzelne Clusterknoten gegen Angriffe geschützt ist. Damit lassen sich mit der TBCA beispielsweise auch Office-PCs in einem Büro, die heutzutage über mehrere Rechenkerne verfügen können, zu einem High-Performance-Cluster verbinden, ohne Sicherheitsrisiken in Kauf nehmen zu müssen. Die einzelnen Clusterknoten müssen dabei nicht alle Rechenkerne zur Verfügung stellen, so dass die Büroarbeit an den PCs weiterhin möglich ist.

Im Bereich Ausfallsicherheit bietet die TBCA die Möglichkeit, die Parameter und den Stand der globalen Variablen mit denen ein ausgelagerter Thread gestartet wird zu speichern. Sollte der Thread nicht korrekt ablaufen, weil beispielsweise der Clusterknoten ausfällt, kann der Thread erneut mit den gleichen Werten gestartet werden. Diese Ausfallsicherheit funktioniert allerdings nur für komplett unabhängige Threads und muss daher in der Konfiguration explizit eingeschalten werden.

5 Beispiel: Threadpool

Die TBCA bietet bereits eine Vereinfachung in der Programmierung eines High-Performance-Clusters. Eine weitere Vereinfachung für Problemstellungen, die eine symmetrische Parallelisierung ermöglichen, ist mit Hilfe eines Threadpools möglich. Durch einen Threadpool wird eine festgelegte Anzahl an Threads bereitgehalten, die anschließend Teile des Gesamtproblems bearbeiten.

```
void run_thread_pool (int numberOfThreads,
void *(*init) (void),
void *(*func) (void *),
void (*post) (void *) );
```

In unserem Beispiel erzeugt die Funktion `run_thread_pool()` `numberOfThreads` Threads zur Bearbeitung des Problems. Als Wert für `numberOfThreads` bietet sich die Anzahl der zur Verfügung stehenden Rechenkernen an. Diese Anzahl wird von der TBCA in der Variablen `tbca_number_of_cores` bereitgestellt.

Für die parallele Bearbeitung eines Problems muss der Programmierer lediglich die drei Funktionen `init()`, `func()` und `post()` implementieren. Die Funktion `init()` stellt für den Durchlauf eines Threads die notwendigen Parameter bereit. Die Funktion `func()` ist die Implementierung der eigentlichen Threadfunktion, die ein Teil des zu berechnenden Problems abarbeitet. Die Funktion `post()` verarbeitet die Ergebnisse eines Threaddurchlaufs.

Die Funktionen `init()` und `post()` müssen keine Rücksicht auf mögliche Konflikte im Zugriff auf gemeinsame Speicherbereiche nehmen. Die notwendigen Sperren werden durch die Funktion `run_thread_pool()` gesetzt. Dadurch werden allerdings die Zeiträume für die Sperren länger als unbedingt notwendig. Bei der Funktion `func()` muss der Entwickler jedoch Einschränkungen beachten. Die Funktion darf nur auf solche globalen Variablen zugreifen, auf die keine andere Threadfunktion zugreift. Denn die Funktion `run_thread_pool()` kann Sperren nur für die gesamte Funktion setzen. Damit wäre eine parallele Arbeit jedoch nicht möglich.

6 Zusammenfassung

Die TBCA ist eine Middleware, mit deren Hilfe eine Thread-basierte Anwendung nicht nur auf einem einzelnen SMP-System sondern auf einem Cluster ablaufen kann. Da ein großer Anteil der Kommunikation zwischen den einzelnen Clusterknoten direkt durch die Applikation erfolgt, ist der Verwaltungsaufwand vergleichsweise gering. Dennoch kann das System eine Applikation an die zur Verfügung stehende Hardware in begrenzten Umfang anpassen.

In der TBCA ist es wie in jedem Rechnernetzwerk notwendig, die zu bearbeitenden Daten über ein Verbindungsnetzwerk zu übertragen. Die Performance einer speicherintensiven Anwendung wird aus diesem Grund durch die TBCA nur gering oder gar nicht gesteigert werden können.

In der derzeitigen Implementierung sind lediglich Semaphore und Mutexe als Synchronisationsmethoden implementiert. Verteilung der Programme auf die einzelnen Clusterknoten erfolgt über NFS-Freigaben auf dem Server. Threaddaufrufe auf Client-Knoten werden derzeit nicht verteilt.

Literatur

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. <http://phase.hpcc.jp/mirrors/mpi/mpich/index.html> (1995)
2. OpenMP Architecture Review Board: OpenMP Application Program Interface. <http://www.openmp.org/drupal/mp-documents/spec25.pdf> (2005)
3. The Open Group: System Interfaces, Issue 6, 2004 Edition. <http://www.opengroup.org/bookstore/catalog/c047.htm> (2004)
4. Al Geist, Adam Beguelin, Jack Dongarra: PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing. Mit Pr (1994)
5. Jan Haase, Frank Eschmann, Bernd Klauer, Klaus Waldschmidt: The SDVM: A Self Distributing Virtual Machine for computer clusters. Lecture Notes in Computer Science, Vol. 2981. Springer Verlag, Berlin Heidelberg New York (2004)
6. Amnon Barak, Oren La'adan: The MOSIX Multicomputer Operating System for High Performance Cluster Computing. Future Generation Computer Systems, Vol. 13. (1998) 361–372
7. Aaftab Munshi: The OpenCL Specification Version 1.0. <http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf> (2008)

On the performance of virtualized cluster computing

Stephan Briskorn, Udo Hönig, and Wolfram Schiffmann

University of Hagen, Department of Mathematics and Computer Science,
Universitätsstr. 1, 58084 Hagen, Germany
{stephan.briskorn, udo.hoenig, wolfram.schiffmann}@fernuni-hagen.de
<http://ra.fernuni-hagen.de>

Abstract. Virtual machines greatly facilitate the setup and operation of infrastructures for parallel computing. Thus, it can be expected that they will become more popular and widespread in the future. In this paper we analyzed a small virtual environment and evaluated the performance reductions due to the virtualization's overhead. The performance loss of virtualization depends mainly on the deployed hypervisor and applications. For the XEN-hypervisor and communication intensive MPI-benchmarks, we found out that there can be an overhead due to virtualization of up to 27 %. In order to lower this overhead further investigations must be carried out to identify and reduce virtualization bottlenecks.

Key words: Parallel computing, MPI, virtual machines, hypervisor, performance evaluation

1 Introduction

Nowadays parallel computing is either based on shared memory or message passing computer architectures. Shared memory architectures are essentially provided by hardware systems with multicore processors which are already supported by modern operating systems. On the other hand, the setup of a compute cluster for message passing programming is pretty complicated because multiple copies of the operating system must be installed and maintained on all cluster nodes. Moreover, the nodes must be connected by fast and low latency network links and configured in such a way that they are addressable in the cluster's local network.

An attractive alternative to running an own cluster is the use of *cloud computing*. Instead of using physical computer systems, the cloud computing approach is based on virtual machines [1] which will be accessed and controlled via the internet. These VMs are rent by the hour from a cloud computing provider. Thus, we do not have to buy and run own real hardware but we get the needed compute power on demand and pay for it by means of a credit card. For example, if we need a big cluster we could use Amazon's Elastic Cloud Computing (EC2) facility to dynamically allocate the needed resources [2]. In order to consolidate the installed physical server systems EC2 uses the virtual machine approach which also allows the users to save and restore the current status of the virtualized computer systems. In this way, the computations can be easily checkpointed at any time by means of a machine image and resumed later on from that status.

If we want to benefit from cloud computing for facilitating parallel processing, the following question arises: To which extent does virtualization degrade the performance of a parallel program? In order to answer this question, we analyzed the runtimes of three parallel benchmark programs both on a small physical cluster and on an equivalent cluster that was virtualized by means of a XEN-hypervisor on the same physical hardware. The usual software tools to implement parallel programs for message passing architectures are MPI and PVM. With these tools one can write portable parallel applications. A great variety of standard High Level Languages e.g. C is supported by those programming interfaces. In order to analyze the performance loss due to virtualization we selected three MPI-based benchmark programs that require a high amount of communication between the compute nodes. In this way, we hopefully evaluated the worst case of the performance reduction by using virtual instead of physical machines.

The outline of the paper is as follows. The next section provides an overview of related work in three categories: parallel programs and architectures, virtualization and virtual machines, and the intersection of these two areas which represents virtualized parallel computer environments. Next, we describe the experimental setup and discuss the results of our investigations, which are finally summarized by a conclusion.

2 Background and related works

The work presented in this paper deals with the effects of virtualization techniques on the execution of parallel programs. In this context, the section at hand will first describe the basics of parallel programs and virtualization, before it discusses related work, that is concerned with the research areas of virtualization and parallel programming as well.

2.1 Parallel Programs and Architectures

In the course of the past decades, an increasing demand for more computing power promoted the development of several generations of new computer systems with continually higher performance. Nevertheless, even today numerous fields of application require more computing power than can be provided by state-of-the-art single-processor architectures. Examples are computer-generated movies and the weather forecasts. Since a further increase of computational speed can not be achieved by a further increase of the clock rate because of physical limitations, parallel computing significantly gains in importance.

By subdividing a programs workload over several cores, processors or computer systems (we will subsume these alternatives by using the term *Processing Element (PE)* throughout the rest of this paper), the execution of a program can usually be accelerated in comparison to sequential processing using a single PE. Parallel Programs consist of multiple tasks that usually depend on each other. A scheduler has to consider all of the data dependencies existing between the program's tasks in order to produce a valid schedule. Most schedulers try to optimize the resulting schedules regarding the minimization of the so called schedule length, which is the time passing between the start of the first and the completion of the last task.

Although many approaches for the development of parallel programs have been proposed over the last few decades, most of them can be reduced to one of the following four basic approaches, namely parallelizing compilers, additional software layers for handling parallelization and execution automatically, dedicated parallel/concurrent programming languages and extensions of existing sequential programming languages. Today, most parallel programs are developed by means of commodity programming languages, e.g. C, C++ or Fortran, which were expanded by additional libraries comprising all constructs needed for parallel programming. The majority of those libraries follow either the PVM [3] or the MPI [4] standard.

In order to process a parallel program efficiently, an adequate hardware infrastructure is required, where the PEs are connected by communication channels with sufficient performance. For this purpose, a broad range of diverse parallel computer architectures has been developed over the years, differing in both, computing power and architectural properties. The majority of recent parallel computer architectures belongs to the classes of multicore processors, multiprocessor systems, clusters and grids. While multicore processors combine at least two independent cores into a single processor, a multiprocessor system possesses two or more processors, which are typically installed on the same mainboard. Both architectures can be classified as shared memory systems, because all available cores or processors can get access to a shared part of RAM in order to achieve a very fast data exchange. In

contrast, clusters and grid systems are composed of several independent computers that are connected by either a local area or wide area network. Both architectures require that data transfers are realized by means of messages that are transferred over the network. A cluster computer typically implements a uniform administration policy on all participating computing nodes causing similar or even identical hard- and software configurations. The individual computers are connected by a high-performance local area network, e.g. Gigabit Ethernet or Myrinet. In grid systems, the participating computing nodes are usually highly heterogeneous concerning their computing power and their hardware equipment. Grid resources are normally provided by different owners, who can all implement their own administration policies, separating the grid into self-sufficient administration domains. In most grid systems, wide area communication is based on the internet.

A comprehensive description of the development of parallel programs and the different kinds of parallel computer architectures can be found in a book by Wilkinson and Allen [5]. Further details on grid systems were summarized by Foster, Kesselman and Tuecke [6].

2.2 Virtualization and Virtual Machines

Modern computer systems comprise a large number of hardware, software and firmware components whose interaction is usually structured and described by means of a layered model. Virtualization introduces a new intermediate layer to a computer's layered system structure that abstracts from the subjacent layers by simulating one or more machines that can be either functionally equivalent or (even completely) different to the underlying platform. These simulated machines, that are typically optimized regarding their performance, are called *Virtual Machines (VM)* and the simulator software is called *Virtual Machine Monitor (VMM)* [7]. As a synonym to VMM, the term *hypervisor* is widely used as well.

While the adaption of a system environment to a user's needs is normally achieved by inserting a new layer on top of the already existing system platform, the new abstraction layer is inserted among the other layers [8], whereas its precise location determines the type of the resulting VMs. Smith and Nair [1] provide a detailed description of several types of VMs, including System VMs, Process VMs, Hosted VMs and Native VMs. In comparison to a modern operating system, a VMM shows a relatively simple structure and small set of features, causing that a VMM is usually smaller than an operating system by orders of magnitude. While the simulated system platform is named *guest*, the system platform that runs the VMM is called *host*. Very often, a single host can process several guests simultaneously. A comprehensive description of the techniques and concepts used in the context of virtual machines can be found in [1] and [9].

Until now, a large number of different VMMs has been developed, enabling the implementation of possibly diverse VMs, that can be different regarding their hosts' and guests' processors and operating systems, their performance and their hardware support, e.g. USB and 3D-accelerators. VMMs that are widely used in the field of x86-based systems are the products of the market leader VMware [10] as well as the virtualization solutions Xen [11], Virtual Box [12] and OpenVZ [13].

In addition to numerous other areas of application, virtual machines can be used particularly to consolidate several servers on a single hardware, to improve a systems' security and to enhance the mobility of users [14]. Moreover, virtualization techniques are applied to simplify the management of a system's hard- and software, enhance the development and test of new software, train employees and customers, improve a system's reliability and to secure the installed software as well as the stored data.

In the context of an improved software development, VMs enable the testing of network software by using only one physical system [7]. In grid environments, VMs can also be

used to implement a mutual protection of resource owners and grid users, to conceal the heterogeneity of hardware resources and software installations, and to generate a flexible virtual network infrastructure [15].

2.3 Virtualization in parallel computer environments

Virtualization simplifies the use of many techniques developed for distributed computer systems during the last two decades of the twentieth century, e.g. migration, checkpointing, load balancing and redundancy, and it also led to a lot of new activities and developments in this area of research [8]. In Grid environments, virtualization is able to improve the flexibility and isolation of the resources' utilization, which is an advantage for both sides, the resources' owners as well as the resources' users. The additional abstraction layer provided by every VMM is able to hide the heterogeneity of the resources. Using virtualization techniques in a grid system will therefore also enhance the security within the grid, the adaptability to the users needs, the utilization of legacy applications, administration and resource management, and the independence of the target system where the grid job is finally executed [16].

The concept of virtualization is not restricted to single computer systems or programs; it can also be applied to independent storage devices like Storage Area Networks. It can be expected, that network resources like routers and switches will be virtualized in the future as well [17]. In particular, a combined utilization of virtual machines and overlay networks can get over the problems implied by the spreading of a grid's resources over several administration domains, separated by firewall and managed by implementing different policies. Examples of systems, that already apply virtualization techniques to achieve standardized grid resources across the borders of administration domains include the WOW-System [15], Instant Grid [18], Nova [19] and VMPlants [20], where the resources are connected by means of a virtual network technology called VNET [21].

In the area of High Performance Computing (HPC), Cluster Computers are a widely used architecture. These systems' computing power and complexity have increased continuously over the years. This growth of complexity has complicated the administrators' task, because every new security update, library and program binary has to be installed consistently on an increasing number of computing nodes [22]. In such environments, virtualization facilitates cloning the individual nodes from a standard node, so that all changes have to be entered only once. Furthermore, the active VMs can be migrated within the virtual cluster, enabling the possibility to remove computing nodes with an emerging malfunction from the cluster without any interference of the remaining system [22].

The above-mentioned benefits of virtualization are paid with an additional overhead, that has to be notably low in a HPC environment. Notwithstanding the obvious advantages, virtual machine technology is thus used only on a limited scale in HPC systems [23]. As a consequence, the number of publications dealing with the development or analysis of parallel programs in virtual infrastructures is pretty small as well. Rezmerita et al. [18] use MPI applications only for analyzing their newly developed middleware, and do not consider the particular aspects of parallel programs in virtualized systems. Emeneker and Stanzione [22] examine the Xen hypervisor and User Mode Linux regarding their performance and reliability on HPC cluster computers by conducting several experiments using HPC Challenge (HPCC) and MppTest as benchmarks. Tikotekar et al. [24] in turn analyze the impact of Xen on the performance of two typical application benchmark suites, HPCC and NPB, and determine in how far it is possible to draw conclusions on the overall overhead from the obtained results. This work's focal point is not on a detailed performance analysis of Xen but on showing an evidence, that results regarding the overhead of a virtual machine monitor can not be generalized if they are based on just one single benchmark suite.

For our best knowledge, the first elaborate examination on the impact of virtual architectures on the runtime of parallel programs can be found in Walters et al. [23]. By using various standard benchmark applications, the authors evaluate several VMMs regarding the processing of computationally intensive HPC applications. The authors first examine the differences between some computer components, i.e. network and file system, of physical and virtualized computer systems regarding their performance. Afterwards they compare the runtime of parallel programs on both systems for shared-memory- as well as for message-passing-programming. In contrast to that approach, i.e. analyzing the architectures bottom-up beginning with the components, we chose to compare the end-to-end runtime of both systems first. This top-down-approach simplifies the identification of interesting issues that should be inspected and focused. Moreover we omit a comparison of shared-memory-programming, as most VMMs use multicore-features internally and thus do not support shared memory programming between several VMs. In addition we take into account, that several VMs might be running on the same host using network means for communication. That scenario gains significant performance advantages, as the network communication is handled within a host's memory.

3 Approach

To accomplish a detailed analysis of the parallel programs' performance in real and virtual system architectures a test environment is required that enables the programs' processing on a real system, while it also allows them to be processed in virtual machines being located on either one or distributed on several computer systems. In order to achieve results of high relevance we selected standard hardware and software components for implementation of the test environment. The hardware basis for all experiments is a dedicated Beowulf cluster consisting of two identically constructed Personal Computers with Intel Core 2 Duo CPUs (E6750), 4 GB RAM and 320 GB hard disk that are connected by a Fast Ethernet network. Both computers are installed from scratch to avoid any relicts from earlier installations or configurations which can affect the findings of our investigations. All results presented in section 4 do not consider hardware virtualization or hardware partitioning, since the former belongs to the standard of new PCs and the last-mentioned comprises no performance-relevant components. Instead, the focus of our analysis is on software virtualization. Due to the market dominance of System Virtual Machines we will concentrate our work on this kind of VM by using the open-source hypervisor Xen 3.1.3 as basis of our investigations.

As was already mentioned in section 2, recent parallel programs are usually implemented by means of commodity imperative or object-oriented programming languages, e.g. C, C++ or Fortran, which were expanded by additional libraries comprising all constructs needed for parallel programming. Therefore, we also followed this approach by utilizing the MPI-standard which in the meantime is more up-to-date and widespread than PVM. Out of the existing MPI-implementations, we used the MPICH derivative from Argonne National Laboratory as well as OpenMPI, which was developed by several research facilities in collaboration with commercial enterprises. Both software packets are widely used and free of charge.

Since both MPI-implementations were developed primarily for Unix-like operating systems, it is obvious to use Unix or Linux respectively as operating system for all host and guest platforms. Because all common versions of Linux are based on the same kernel sources, there is no compelling exclusion criterion for any of the widespread distributions. For this reason, we decided to use the 10.3 release of the openSUSE distribution that was the latest version when we started our work. This selection was taken mainly for two reasons: On the one hand, openSUSE is a popular open source branch of the widely-used commercial

SUSE-Linux distribution. On the other hand, openSUSE 10.3 contains a preconfigured Xen-capable kernel, whose configuration and integration is already optimized by the developers of the distribution.

Each of the conducted investigations was intended to consider the overall virtual environment, making benchmarks that are biased to a single resource (like the CPU) unsuitable for our approach. Additionally, in order to get meaningful results, we had to look for benchmarks which

- consider the time required for initializing the communication path,
- are able to guarantee consistent results regardless of other active processes,
- admit only an insignificant impact on the network’s performance and that
- consider the accuracy of the computer system’s clock.

mpptest and Spezial Karlsruher MPI-Benchmark (SKaMPI) are two well-established and approved benchmarks that possess all of the mentioned properties and that can also be used to compare different system platforms by using the same MPI-implementation.

The accomplished analyses require the installation of two Xen-VMs on each of the two PCs. Thus, it is possible to measure both the performance that can be obtained with two guests on the same PC as well as the performance of two communicating VMs that are placed on different computers. The investigations concerning the real system platforms are carried out on the basis of the host platform. Each VM is equipped with one CPU, 512 MB RAM and a bridged network interface. Since our analyses require absolutely identical VMs, the virtual hard disks are stored as disk images in the hosts’ file systems enabling a quick and easy cloning of a preconfigured VM at the expense of a slightly reduced performance. In order to avoid further performance losses, the virtual disks are allocated with their maximum size right at the beginning. The software installed on both hosts and guests was reduced to the absolutely necessary minimum.

4 Results

To gain exploitable results it is necessary to execute each benchmark not only once per each configuration, but to gather a set of results enabling the analyst to eliminate results which are obviously influenced by side-effects. Although the environment is built from scratch the results show up that not all kinds of side-effects can be eliminated technically. Therefore each benchmark is executed 50 times per configuration. Looking at a list of results they converge to the lower limit of the runtime of the particular configuration pointing out the minimum. This value as target of each performance optimization will further on be regarded as reference for each configuration and hence will be the basis for the comparison of the technologies. As first benchmark the MPDRingtest is applied, which is part of the MPICH package and therefore can only serve to compare physical and virtual platforms. MPDRingtest builds a ring through all involved stations sending a package n times through this ring and measures the time the package needs to pass all stations and reach its origin the n^{th} time. With the virtual platforms we have to distinguish between 2 VMs running on one host and communicating locally and two VMs running on two separate hosts bridging a longer distance in their communications. Figure 1 summarizes the results of MPDRingtest. The benchmark affirms the expectation that the VMs perform more slowly than the physical systems. The virtualization overhead is approximately 27%. As expected, the values also improve significantly, when the distance between the VMs is reduced by running them on the same host. That configuration enables the VMs to perform the benchmark within a third of baseline of the physical machines. This simple experiment shows up the tremendous significance of the network distance on the performance of a cluster.

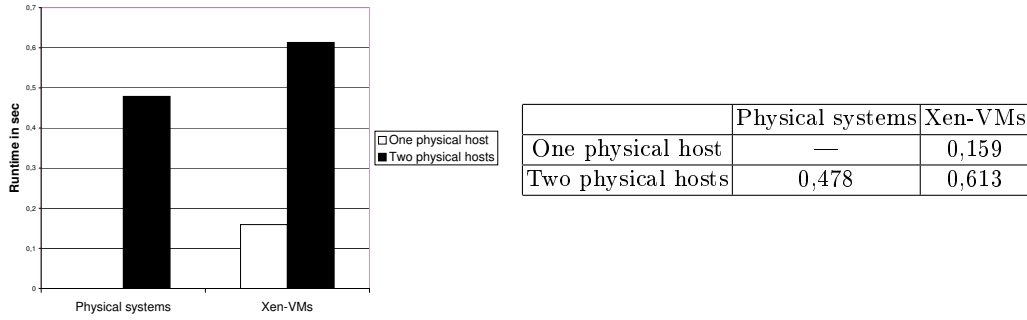


Fig. 1. Overview of MPDRingtest results (Runtime in seconds)

In contrast to MPDRingtest the mpptest-benchmark sends a sequence of messages of variable length between the cluster partners. The summary is shown in figure 2. The benchmark allows a more differentiated view as it uses larger packets and is executable with OpenMPI.

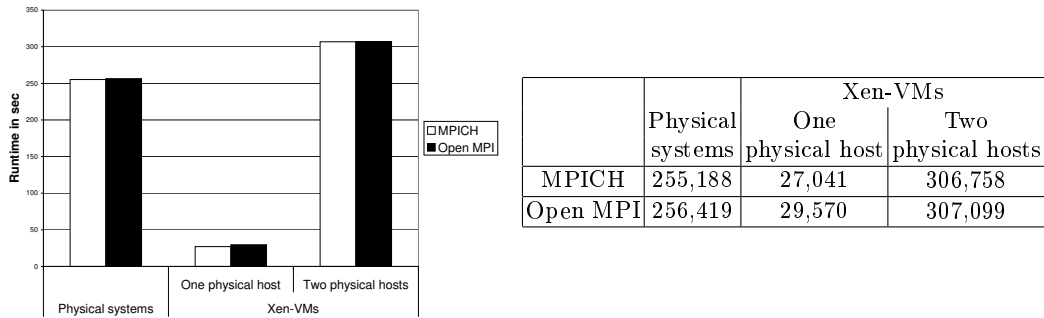


Fig. 2. Overview of mpptest results.

Similar to MPDRingtest you also find the VMs performing 20% slower than the physical systems unless they are running on the same host when they take only 11% of the time. These values indicate that the relative VMM overhead shrinks due to increasing message length. Following this assumption it is likely that common parallel algorithms cause relatively less virtual overhead than the benchmarks, that therefore reflect the worst case. There is no significant difference between the two MPI-implementations.

Eventually we have a look at the SKaMPI benchmark. It sends a sequence of messages between the cluster partners using different send-procedures. Our results are summarized in figure 3.

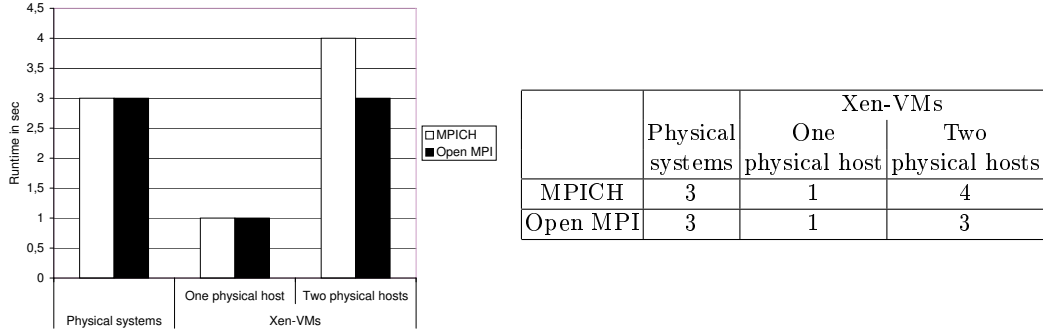


Fig. 3. Overview of SKaMPI results.

It underlines the results so far. Some irritation is caused by the performance gains of OpenMPI in relation to MPICH on the Xen-VMs. Going beyond the rounded results and taking a closer look on the detailed results of the execution with one physical host it becomes obvious that OpenMPI performs better in that configuration as well. The main difference in the execution time is caused by send procedure 2, which is the only one using the MPI_Iprobe-function, which checks incoming messages before processing them.

Coming back to the basic question of the performance overhead of virtual machines executing parallel programs a comparison between the physical machines and the best VM-configuration is pointed out by figure 4.

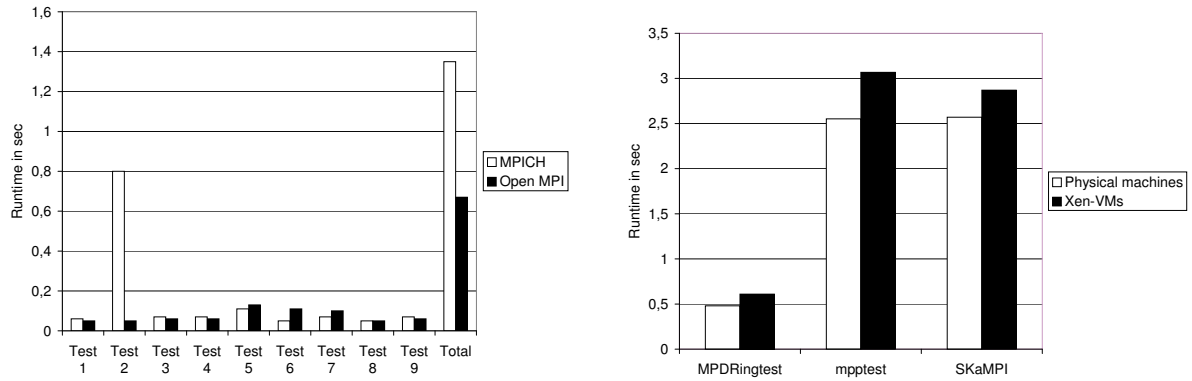


Fig. 4. Left: Synopsis of MPICH and OpenMPI in each send procedure with Xen. Right: physical machines and best VM-configuration.

The overhead is approx. 27% in MPDRingtest, 20% in mpptest and 12% in SKaMPI.

5 Conclusion

In the future we expect that parallel computing will move to virtualized platforms because they are easier to install and maintain. Thus, the question ‘To which extent does virtualization degrade the performance of a parallel program?’ will become more and more important.

In this paper we analyzed a small virtual environment and presented measurements that help to answer this question. The influence of virtualization must be evaluated with respect to the used hypervisor *and* the specific applications. For the XEN-hypervisor, we found out that there can be an overhead due to virtualization of up to 27 %. Fortunately, this pretty huge overhead can be explained by the high portion of network traffic in the chosen benchmarks. Thus, we guess that the measured values reflect the worst case and that we will hopefully get lower overheads in case of practical applications. We will try to verify this hypothesis in our future work. Also we will investigate the performance of commercial virtual machines in combination with embarrassingly parallel applications.

References

1. Smith, J.E. and Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005
2. <http://aws.amazon.com/ec2/>
3. Geist, A., et al.: PVM 3 Users Guide and Reference Manual. Oak Ridge National Laboratory, 1993
4. Gropp, W., Lusk, E., Skjellum A.: Using MPI: portable parallel programming with the message-passing interface, The MIT Press, 1999
5. Wilkinson, B. and Allen, M.: Parallel Programming – Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice Hall, 2005
6. Foster, I., Kesselman, C. und Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In: International Journal of Supercomputer Applications, Vol. 15, No. 3, pp. 200 – 222, 2001.
7. Goldberg, R.P.: Survey of Virtual Machine Research. Computer, Vol. 7 , No. 6, pp. 34 – 45, IEEE (1974)
8. Figueiredo, R., Dinda, P.A. and Fortes, J.: Resource Virtualization Renaissance. Computer, pp. 28 – 31, IEEE Computer Society (2005)
9. Smith, J.E. and Nair, R.: The Architecture of Virtual Machines. Computer, pp.32 – 38, IEEE Computer Society (2005)
10. <http://www.vmware.com/>
11. Barham, P. et al.: Xen and the Art of Virtualization, In: SOSP'03: Proceedings of the nineteenth ACM Symposium on Operating System Principles, pp. 163 – 177; New York, NY, USA; ACM Press, 2003
12. <http://www.virtualbox.org/>
13. <http://wiki.openvz.org/>
14. Chen, P.M. and Noble, B.D.: When Virtual Is Better Than Real. HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, pp. 133 – 138, IEEE Computer Society (2001)
15. Ganguly, A., et al.: WOW: Self-organizing Wide Area Overlay Networks of Virtual Workstations. Journal of Grid Computing, Vol. 5, No. 2, Springer (2007)
16. Figueiredo, R.J., Dinda, P.A. und Fortes, J.A.B.: A Case for Grid Computing on Virtual Machines, In: Proceedings of the. 23rd International Conference on Distributed Computing Systems, pp. 550 – 559, IEEE Computer Society, 2003
17. Rosenblum, M. and Garfinkel, T.: Virtual Machine Monitors: Current Technology and Future Trends. Computer, pp. 39 – 47, IEEE Computer Society (2005)
18. Rezmerita, A., et al.: Private Virtual Cluster: Infrastructure and Protocol for Instant Grids, In: Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Lecture Notes in Computer Science, LNCS 4128, pp. 393 – 404, Springer-Verlag Berlin Heidelberg, 2006
19. Sundarrajan, S. et al.: Nova: An Approach to On-Demand Virtual Execution Environment for Grids, In: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), IEEE Computer Society, 2006
20. Krsul, I., et al.: VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing, In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, pp. 7 – 18, IEEE Computer Society, 2004
21. Sundararaj, A.I. and Dinda, P.A.: Towards virtual networks for virtual machine grid computing, In: VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium, pp. 14 – 27, USENIX Association, Berkeley, CA, USA, 2004
22. Emeneker, W. and Stanzone, D.: HPC Cluster Readiness of Xen and User Mode Linux, In: IEEE International Conference on Cluster Computing, pp. 1 – 8, IEEE Computer Society, 2006
23. Walters, J.P. et al.: A Comparison of Virtualization Technologies for HPC, In: Proceedings of the 22nd Conference on Advanced Information Networking and Applications, pp. 861 – 868, IEEE Computer Society, 2008
24. Tikotekar, A., et al.: An Analysis of HPC Benchmark Applications in Virtual Machine Environments, In: Lecture Notes in Computer Science: Proceedings of the 14th European Conference on Parallel and Distributed Computing (Euro-Par) 2008: 3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing (VHPC) 2008, Las Palmas de Gran Canaria, Spain, August 26-29, 2008. Springer Verlag, Berlin, Germany, 2008

Providing Guidance Information for Application-Mapping on Heterogeneous Parallel Systems

Fabian Nowak, Rainer Buchty, and Mario Kicherer

Chair for Computer Architecture,
Universität Karlsruhe (TH)
P.O.Box 76128 Karlsruhe, Germany
{nowak|buchty|kicherer}@ira.uka.de
<http://ca.itec.uka.de>

Abstract. With the advent of heterogeneous parallel systems, a method is required for scheduling and mapping applications on the available hardware. We propose the concept of attributes enabling a runtime system to decide which implementation is usable, given certain constraints such as available hardware resources or power usage. The proposed approach does not break compatibility with existing source code and binary files and can be tightly integrated into a runtime system handling these additional specifications and allowing for reconfigurable hardware to be used according to application requirements and system configuration.

Key words: Multi-Core, Reconfiguration, Scheduling, Attributes

1 Introduction

Exploiting Moore's law, multi-core and many-core architectures are the current trend in processor design. With these architectures, performance comes mainly from the number and type of processing units rather than a high clock rate as with past uniprocessor cores. Continuing the multi-core trend, custom modules for dedicated purposes and reconfigurable systems are becoming more and more attractive and are being built into high-performance, desktop, and embedded systems.

In order to successfully establish heterogeneous environments in commodity systems, additional means have to be found that allow employing the available resources to the maximum extent that is sensible and permissible by internal and external constraints; an example is provided in [6]: for the given example, an appropriate decision whether to choose a hardware or software implementation therefore requires knowledge of the runtime behaviour associated with data size.

Most gain of such systems can be obtained when offering different function implementations for different hardware resources so that according to constraints and requirements as mentioned before, the most suitable execution unit can be chosen. We already proposed a mechanism for mapping function calls onto different implementations [1] using a lightweight runtime system that basically inserts a virtual layer between the application and the hardware resources. As of now, this runtime system does not use an automatic way of workload assignment; the decision-making of when to execute which function implementation on which unit has to be supported explicitly.

Using so-called attributes, this paper presents an approach to support the process of scheduling and mapping the functions and their implementations for different resources on heterogeneous multi-core systems on single hosts. Attributes pose a suitable means to notify our runtime system of an application's requirements, to track the current system state that is obtainable via monitoring, and to make decisions based on threshold values and rules where to best execute a function call. They can be specified in any source code files and extracted by extensions of the respective compiler tool-chain. Function attributes

are then stored within executable binary files and libraries in the Executable and Linkable Format (ELF), hence enabling convenient access by the runtime system’s mapping guide and keeping binary compatibility to regular execution environments. Knowing attributes both of the calling function and of possible target functions in different repository-like libraries enables scheduling software to chose proper implementations.

The paper is structured as follows: Section 2 discusses the related work, in Section 3 the concept of our approach is then explained together with some trade-offs and specific solutions. The implementation of our approach is presented in Section 4. Afterwards, the integration of the concept into our runtime environment is depicted in Section 5. Section 6 concludes the paper and presents our future work in this area.

2 Related Work

In the domain of heterogeneous and reconfigurable architectures, some work has been accomplished that directly addresses programmability issues: EXOCHI [14] is an environment for execution on heterogeneous multi-core systems where the executable binary is extended by accelerator-specific binary data that can be used to simultaneously execute threads both on accelerators and software; Lime [8] tries to bridge the gap between software and hardware programming by extending Java by hardware-near types. However, scheduling and mapping is precomputed using some work-estimation heuristics, which is not generally applicable to the bigger part of applications and programming flows. Furthermore, the concepts are fixed to one programming model and one language only.

Another static approach to building programs based on function calls for different platforms is meta-programming [13], where models are created and then derived for specific architectures, that can also be employed for heterogeneous systems. Although there are meta-programming approaches that do not only create static mappings but rather dynamically compile for even heterogeneous platforms [11], its focus is still on the code generation itself rather than on ease of programmability and specification of objectives.

Mapping of functions and applications onto systems is frequently achieved via flow graphs as in [9]. Possible solutions of how to map functions onto the available resources based on the evaluated application attributes and system state can be found via multi-objective optimisation, delivering a pareto-optimal schedule of both the resources and CPU time for the running attributed tasks. The scheduling problems are solved for example by Dutta et al. [3] with integer linear programming (ILP) being used as means for multi-objective optimisation. A general concept of where to store application-specific information is missing; also, extensibility by new hardware is not granted.

ASKALON [4] is a grid environment for deploying work flows of scientific applications, addressing resource allocation and reliable and fault-tolerant execution. The applications are specified in an XML-based language. Requests for allocating work flows on resources can contain high-level requirements such as clock rate or operating system. Similarly, the Arches framework focuses at bridging the gap between domain-specific scientists and HPC programmers, but does not target resource allocating and job scheduling [2].

The concept to extend a program with additional information has proven to be successful by OpenMP, where so-called pragmas are used to direct the compiler to parallelise the following code region. Also, annotating program code on a semantic level, indicating for example the role of a class instance, is done by Mügge et al. [7].

In contrast to the afore-mentioned works, our concept aims at supporting scheduling and mapping of applications for heterogeneous systems while keeping source code compatibility and not introducing new languages or programming models. It furthermore enables us to execute both regular executables and binary files extended with attributes.

3 Concept

The overall problem under investigation is how to execute programs with regard to assertions and constraints when a specific, but not fixed, perhaps even adapting and changing hardware/environmental setup is given. In such a setup, scheduling has to be achieved considering fixed constraints such as maximum power usage or minimum runtime in order to adhere to power limitations or deadlines. The problem addressed in this paper is how to support a runtime system in finding suitable schedules while integrating smoothly into today's systems and extending existing concepts and programming flows.

As solution to this problem, we suggest to attribute both programs and libraries and to evaluate these attributes in a mapping guide. This is shown in Fig. 1: An attributed executable binary file is started in the runtime environment, providing its attributes to a mapping guide and having its function calls be directed to suitable implementations in software or hardware. The implementations might refer to hardware frameworks [6] that provide implementations in so-called slots with status interfaces via a file system interface called ProcFS. The mapping guide directs the runtime system which function calls to redirect to which different implementations in order to fulfil requirements and features as specified by the attributes and given by the internal system state and external constraints.

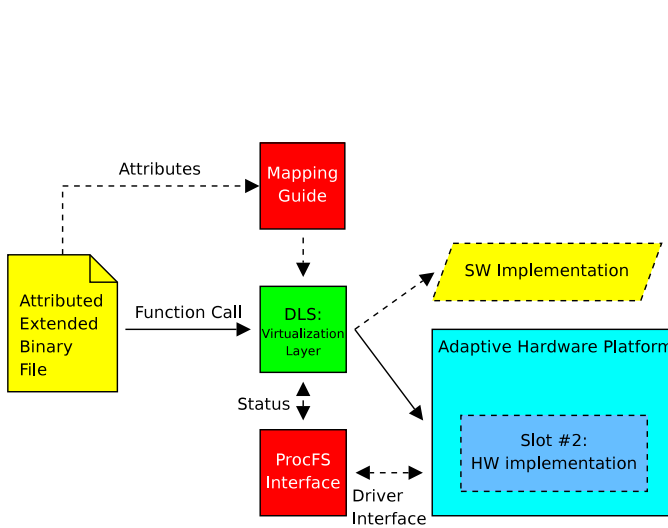


Fig. 1. Overview of the general attribute concept

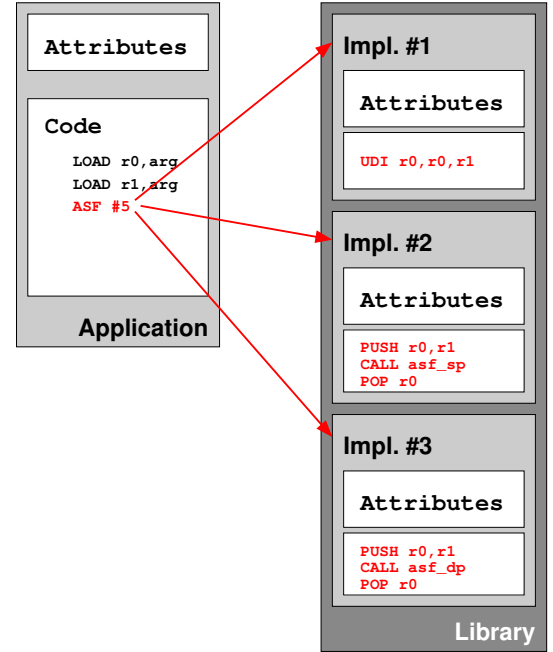


Fig. 2. Mapping of a function onto suitable implementation

It also controls the general system setup, such as controlling reconfiguration of FPGAs or granting exclusive access onto ASIC resources. Inside the virtualization layer of the Dynamic Linking System (DLS), function calls are then routed to different implementations in software or on available hardware according to the commands of the mapping guide.

This is elaborated more precisely in Fig. 2: both the application with the function calls inside and the available software implementations are to be attributed, alleviating the task of deciding which implementation to use.

As already mentioned before, the attributes need to be extracted from the program code via a preprocessor and inserted into the binaries. The flow of attribution, extraction and extension is depicted in Fig. 3 with both the source file and the binary file. First, the

program sources need to be extended by pragmas with the statically known information of program requirements and implementation capabilities. Then, an extended GCC-based toolchain extracts the pragmas, compiles the source code and links the objects, and finally attaches the extracted attributes to the executable. This information is stored in a new DLRS segment, which contains an arbitrary attributes segment among others, as can be seen on the right.

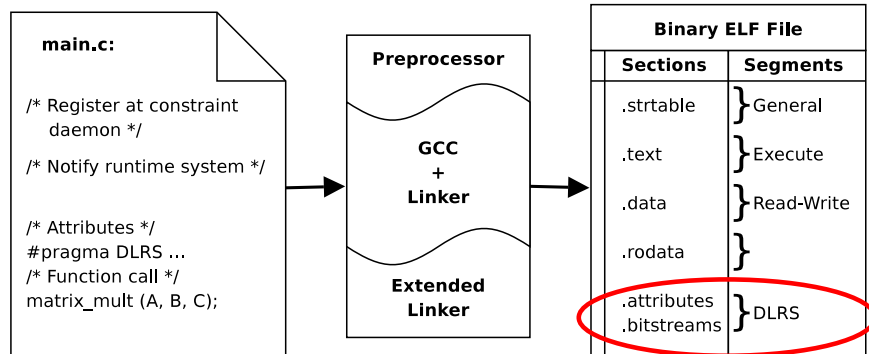


Fig. 3. Attributed Program and Compiled Binary File

With attributes such as bandwidth, maximum memory consumption, preferable input data size and normalised execution times, it is possible to automatically call a suitable implementation because specific demands or properties of software or hardware implementations are known from the attributes of their containing libraries. In the long run, the attributes will help close the feedback loop: applications and monitoring systems can adjust attributes such as the reliability to the measured values, derive new attributes or even indicate incorrect implementations and broken hardware.

In our twofold approach, we propose the use of pragmas to augment the program code with attributes achieving that source code remains compatible with existing programming environments and compilers, and the binary extension is in such a way that annotated binaries can be executed in legacy environments and regular binaries can still be executed in our environment. Also, the general approach is completely language-agnostic. The following subsections explain the extensions in more detail.

3.1 Code Extension

Attributing program code works as follows: both function implementations and calls within the Dynamic Linking and Runtime System (DLRS) are prefixed with `#pragma DLRS` and an arbitrary number of `key=value` pairs that describe the features or requirements of the implementation. In order to have a usable parameter set, the programmer and the surrounding tools are required to use orthogonal parameters only, i.e. they do not depend on or influence each other, and to use the same attribute names for all functions in a project. By integrating these parameter names into existing programming environments, this does not pose any severe restrictions. Secondly, function calls are attributed in the same way, hence specifying the requirements if available. To give a better understanding, Fig. 4 contains two annotated function implementations of the *3DES* function, while in the *main* method the requirements are given.

As can be seen from this example where both implementations seem to equally suit the attributed function call with regard to sensitive data sizes, it is necessary to have additional high-level constraints such as power-efficient execution in order to facilitate the decision which implementation to choose. For example, this can be achieved by classifying hardware

```

#pragma DLRS name=calculate_3DES data_size_min=10KB target=FPGA
double * calculate_3DES_hw (double* x) {
    return hmol_accelerate_direct (accel_slot , ...);
}

#pragma DLRS name=calculate_3DES data_size_max=50KB target=CPU
double * calculate_3DES_sw (double* x) {
    double *encrypted;
    ...
    return encrypted;
}

int main (int argc, char **argv) {
    double *input, *encrypted;
    input = ...
    #pragma DLRS prefer_hardware=1 data_size=30KB
    encrypted = calculate_3DES (input);
    ...
}

```

Fig. 4. Attributed Caller and Callee Implementation

implementations as generally more power-efficient, so `calculate_3DES_hw` would be the choice for power-efficient execution.

3.2 Binary Extension

For the attributes to be used by the runtime system and constraint daemons, we suggest to extend the basic executable file by the attributes. A different approach would be to deliver additional information files together with the executable, but this would harm the basic concept of how programs are delivered and executed.

Executable files for modern UNIX-like systems are stored in the Executable and Linkable Format [10]. Basically, such an executable consists of several headers that specify the number of segments and their offsets in the file. A segment may comprise several sections in the file. Sections contain program code, constants etc.

We propose to add a new segment for the Dynamic Linking Runtime System. As of now, it consists of two sections: attributes and hardware bitstreams. In the first section, all the specified attributes have to be stored for each attributed function of any source file. To access the names, either the regular string section can be used or a new string section within the DLRS segment. The second section for the bitstream is not addressed in this paper.

4 Implementation and Results

4.1 Toolchain

The attributes from the extended code need to be acquired by parts of the programming tool-chain as has already been indicated in Figure 3. We developed a separate preprocessor in Python that is able to extract the DLRS-specific pragmas. It is designed in such a way that pragmas such as these of OpenMP can still be used.

For extending the ELF binary, we developed an ELF editor as depicted in Fig. 5.

4.2 Evaluating Attributes at Runtime

With respect to the already existing runtime system, the question occurs where and how to evaluate attributes.

The evaluation could occur within the application itself, which is not a preferred approach as applications cannot be aware of other programs' resource demands and of the overall

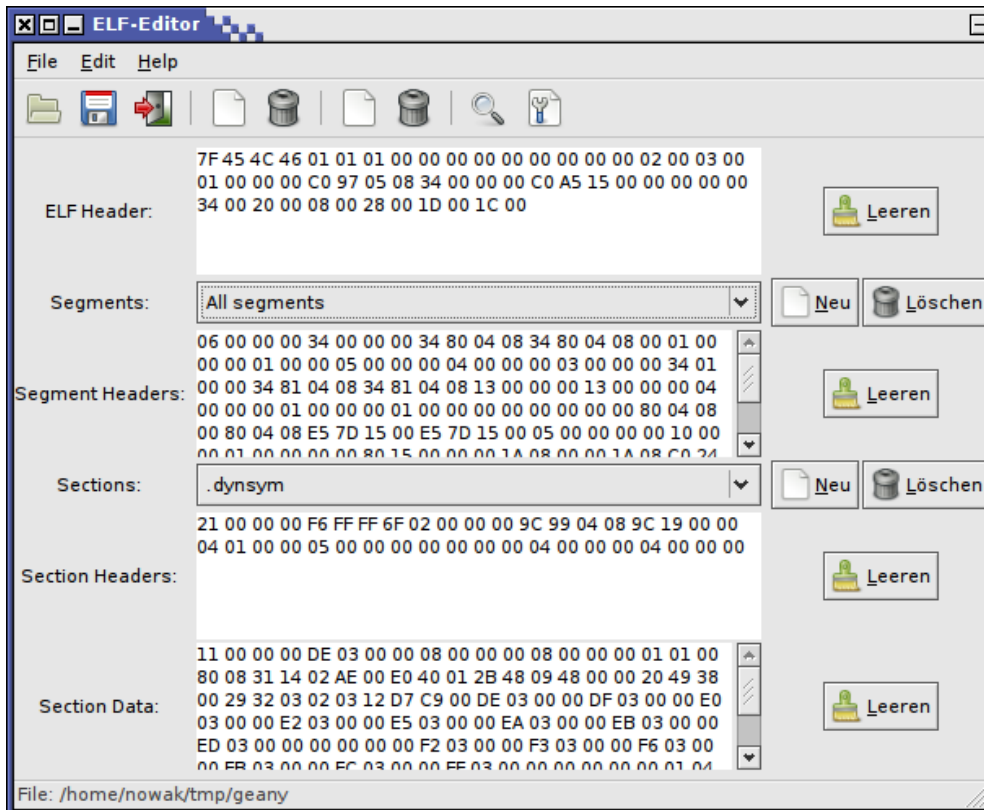


Fig. 5. ELF Editor for working with headers, segments and sections of ELF files

system status. Hence, there has to be a central instance that is aware of the available computing and memory resources in the system and of the remotely accessible resources as well. The application attributes are stored within the binary and hence can be retrieved via ELF access libraries such as libelf [5] from both the application itself and from the central instance. For the central approach, this instance has to be notified of a new “client”. This can be achieved if the client program registers at the instance so that the instance obtains the name of the program’s executable and can extract the attributes. The system attributes are on the one hand obtained from monitoring instances in the system and from those attached to the hardware directly, and on the other hand from the amount of occupied and free resources.

5 Integration into the Runtime System

In this section, we will show how the attribute concept is integrated into our runtime environment [1]. This runtime system [1] leverages switching between different implementations of the same functionality such as executing in hardware or using additional error analysis in contrast to the regular software implementation. The requirement is to allow switching both from within the application and from system level in order to allow more sophisticated schedulers to maximise system usage in terms of number of finished jobs or processed data by employing additional hardware resources such as FPGA boards and graphics adaptors.

There are two possible ways of achieving that: the ELF concept allows to extract function names from an application and upon that, the referenced function implementation can be switched to an alternative one that is already present in the address space. This is called the GOT-based Linking System (GLS). Furthermore, it is possible to reroute the function calls from another application via the Linux kernel. The set of possible function implementations for the function calls can be set up and even extended at runtime, and hence must not be

known at compile time except for one default target. This is our Dynamic Linking System (DLS).

After obtaining the respective notification from a program before the function call finally happens, the mapping guide directs the runtime system to switch the target of a function call to a different implementation that provides additional benefit over the regular one with regard to certain optimisation criteria such as power, performance, or correctness of the result. At this point, the attributes provide the necessary information for deciding which implementation to finally use, adhering to global system constraints.

For executing the program and mapping the program calls, there are two alternatives that both use a central instance as described in 4.2. The first one is calling and executing the program by means of the runtime environment, while the second is introducing a middle layer implemented as a shared library that communicates with the DLS via IPC before each function call to be executed.

The latter is preferable for manually guiding the program. However, we prefer the first alternative because it allows to statically calculate a first mapping of all the program calls and to later on adjust the mapping dynamically.

6 Conclusions and Future Work

In this paper, we presented attributes as an approach for executing different function implementations subject to system and application constraints. The approach requires a preprocessor extracting pragmas that are then inserted by an additional backend into regular executable binary files and into libraries that contain different function implementations. With the introduction of one additional specification layer in addition to the program code itself, the programmer is free to exploit a correspondingly extended system to the maximum extent allowed by the system with as many hardware accelerators in use as possible and sensible and also supporting reconfiguration of reconfigurable resources; but he is still free to use additional programming models such as OpenMP or even other programming languages than C. Attributes proved to be reasonable for making decisions on which implementation to use based on input data size, but can also be used to care for tracking the status of dynamically reconfigurable hardware.

Ongoing work consists of translating high-level user-specific and user-specified attributes into low-level attributes that can be interpreted by the mapping daemon, and of creating metrics for comparing the low-level attributes in such a way that the underlying architecture does not matter. Of special interest are dedicated algorithms for evaluating the attributes.

For the future, we plan to target automatic hardware generation via such pragma attributes (“hardware-synthesizable”) by the preprocessor calling additional tools such as the Trident compiler [12] and other C-to-VHDL tools, aiming at automatic creation of hardware implementation repositories.

References

1. Rainer Buchty, David Kramer, Mario Kicherer, and Wolfgang Karl. A light-weight approach to dynamical runtime linking supporting heterogenous, parallel, and reconfigurable architectures. In *Architecture of Computing Systems – ARCS 2009*, volume 5455 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin / Heidelberg, February 2009.
2. Nathan DeBardeleben, Ron Sass, Daniel Stanzione, Jr., and Walter B. Ligon, III. Building problem-solving environments with the arches framework. *J. Syst. Softw.*, 82(7):1137–1151, 2009.
3. Hritam Dutta, Frank Hannig, and Jürgen Teich. Performance Matching of Hardware Acceleration Engines for Heterogeneous MPSoC Using Modular Performance Analysis. In *Architecture of Computing Systems – ARCS 2009*, volume 5455 of *Lecture Notes in Computer Science*, pages 233–245. Springer Berlin / Heidelberg, February 2009.

4. T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wiczorek. ASKALON: A Grid Application Development and Computing Environment. In *6th International Workshop on Grid Computing (Grid 2005)*, Seattle, USA, November 2005. IEEE Computer Society Press.
5. Joseph Koshy. libelf by Example. <http://people.freebsd.org/~jkoshy/download/libelf/article.html>.
6. David Kramer, Thorsten Vogel, Rainer Buchty, Fabian Nowak, and Wolfgang Karl. A general purpose HyperTransport-based Application Accelerator Framework. In *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA2009)*, pages 20–29. Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg, February 2009. ISBN 978-3-00-027249-3.
7. Holger Mügge, Tobias Rho, and Armin B. Cremers. Integrating aspect-orientation and structural annotations to support adaptive middleware. In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 9–14, New York, NY, USA, 2007. ACM.
8. Shan Huang and Amir Hormati and David Bacon and Rodric Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *Proceedings of the 2008 European Conference on Object-Oriented Programming (ECOOP)*, July 2008.
9. Lodewijk T. Smit, Gerard J.M. Smit, Johann L. Hurink, Hajo Broersma, Daniël Paulusma, and Pascal T. Wolkotte. Run-time assignment of tasks to multiple heterogeneous processors. In *5TH PROGRESS Symposium on Embedded Systems*, pages 185–192. STW Technology Foundation, 2004.
10. The Santa Cruz Operation, Inc. System V Application Binary Interface (Edition 4.1), 1997. <http://www.caldera.com/developers/devspecs/gabi41.pdf>.
11. Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):1–40, 2008.
12. J.L. Tripp, K.D. Peterson, C. Ahrens, J.D. Poznanovic, and M.B. Gokhale. Trident: an FPGA compiler framework for floating-point algorithms. *Field Programmable Logic and Applications, 2005. International Conference on*, pages 317–322, August 2005.
13. Salvador Trujillo, Maider Azanza, and Oscar Diaz. Generative Metaprogramming. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 105–114, New York, NY, USA, 2007. ACM.
14. Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. *SIGPLAN Not.*, 42(6):156–166, 2007.

22. PARS-Workshop (Poster)

Seite

Ansätze zur Parallelisierung eines Verfahrens zur Online-Fehlerdiagnose	125
<i>Alexandru Paler</i>	
GPU accelerated scientific computing: Fluid and particulate flows with CUDA	135
<i>Tobias Hahn, Vincent Heuveline, Björn Rocker</i>	
A Universal Framework for Simulating Hierarchical Network Topologies in a Distributed Memory System	145
<i>Oliver Mattes, Rainer Buchty</i>	

Ansätze zur Parallelisierung eines Verfahrens zur Online-Fehlerdiagnose

Alexandru Paler

Fachhochschule Wiesbaden,
Labor für Verteilte Systeme,
Fachbereich Design Informatik Medien,
Kurt-Schumacher-Ring 18, D-65197 Wiesbaden
alexandrupaler@gmail.com

Zusammenfassung Das Ziel dieser Arbeit ist der Vergleich mehrerer Ansätze zur Parallelisierung eines Verfahrens zur Online-Fehlerdiagnose. Durch ihre lokal approximierende Funktionalität sind neuronale Netze, die auf Radial Basis Funktionen basieren, ein interessanter Ansatz, um nicht klassifizierbare Fehler nicht falsch zu diagnostizieren, und werden als Grundlage eines solchen Systems verwendet. Das inhärent parallele Verhalten von Radial-Basis-Function-Netzen (RBF-Netzen) erlaubt die Analyse möglicher Ansätze für die parallele Online-Fehlerdiagnose. Eine genauere Betrachtung des Aufwandes, der von der Kommunikation induziert wird, erfolgt mit abgeleiteten mathematischen Modellen. Die Modelle liefern ein tieferes Verständnis der möglichen Kommunikations-Problematik, erlauben es, den Kommunikationsaufwand abzuschätzen und das Potentials der Parallelisierungs-Ansätze frühzeitig zu bestimmen. Der Entwurf eines Load-Balancing-Mechanismus, um die Leistung des Fehlerdiagnose-Systems durch exzessive Kommunikation nicht negativ zu beeinflussen, ist eine zukünftige Erweiterung. Der Einsatz der Ansätze in realen Fehlerdiagnose-Systemen wird für möglich gehalten.

Key words: Parallel Programming, Online-Fehlerdiagnose, RBF-Netz, Speed-Up, Performance-Vorhersage des Kommunikationsaufwandes

1 Einführung

Mit der Überwachung technischer Prozesse beschäftigen sich Fault Detection and Diagnosis Systeme (FDD-Systeme). Diese sind für das frühzeitige Erkennen von fehlerhaften Zuständen und Ereignissen zuständig. Eine ausführliche Betrachtung dieses Bereichs erfolgt in [5].

Ansätze zur Fehlerdiagnose, die auf neuronalen Netzen basieren, bieten Vor- und Nachteile. Zu den Vorteilen zählt, dass die Prozess- und Fehler-Modelle nicht unbedingt bekannt sein müssen, was als eine Vereinfachung der Design-Phase gilt. Als Nachteil gilt, dass neuronale Netze als Diskriminatoren und nicht als Klassifikatoren fungieren [1]. Während der sogenannten Trainings-Phase lernen neuronale Netze die Modelle der Prozesse und der Fehler. Im Laufe der Simulations-Phase können die trainierten neuronalen Netze Fehler erkennen und diagnostizieren.

Das Online-Verhalten eines Fehlerdiagnosesystems bezieht sich auf die Geschwindigkeit der Fehlerdiagnose: Fehler müssen schnell diagnostiziert werden. Das zeitliche Verhalten ist nicht streng geregelt, doch muss es sich in bestimmten Schranken bewegen. Online bedeutet auch, dass die gesamte Funktionalität eines Fehlerdiagnose-Systems ohne menschlichen Eingriff statt finden muss.

Die Arbeit gliedert sich wie folgt. Im ersten Abschnitt wird eine kurze Perspektive über RBF-Netze und ihre Eignung für die Fehlerdiagnose gezeigt. Dann werden die Parallelisierungs-Ansätze für RBF-Netze präsentiert und mathematisch modelliert. Die Umsetzung und die Ergebnisse der Parallelisierung werden in den zwei vorletzten Abschnitten beschrieben. Im letzten Abschnitt wird die Arbeit zusammengefasst und ein Ausblick über weitere Erweiterungen gegeben.

2 Fehlerdiagnose mit RBF-Netzen

Neuronale Netze werden ausführlich in [2] beschrieben. RBF-Netze sind eine besondere Art von neuronalen Netzen. In [2] werden folgende grundlegende Definitionen erläutert:

- Das RBF-Zentrum μ ist ein Punkt im Eingabe-Raum.
- Ein RBF-Neuron besitzt eine Propagierungsfunktion, welche den Abstand zwischen dem RBF-Zentrum μ eines Neurons und dem Eingabevektor in feststellt. Die Radial Basis Function φ ist die Aktivierungsfunktion.
- Ein RBF-Ausgabe-Neuron besitzt die gewichtete Summe als Propagierungsfunktion und die Identität als Aktivierungsfunktion. Das bedeutet, dass die Ausgabe-Neuronen die Eingaben aufsummieren und die Summe ausgeben.
- Ein RBF-Netz besitzt genau drei Schichten. Jede Schicht ist mit der nächsten voll verknüpft, also wird eine Feed Forward-Architektur gebildet. Nur die Verbindungen zwischen den RBF-Neuronen und den Ausgabe-Neuronen sind gewichtet. Die Gewichte werden mit w_{ij} für $i = 1, n$ und $j = 1, g$ bezeichnet. Die Menge der Eingabe-Neuronen wird mit I ($d =_{\text{def}} \#I$), die Menge der RBF-Neuronen mit G ($g =_{\text{def}} \#G$) und die Menge der Ausgabe-Neuronen mit O ($n =_{\text{def}} \#O$) bezeichnet.

RBF-Netze können als Durchflussgraphen dargestellt werden, und das ähnelt der in [2] beschriebenen Vorgehensweise. Der Vorteil einer solchen Darstellung liegt in der besseren Veranschaulichung der Operationen die von einem Netz ausgeführt werden. Eine genauere Beschreibung der Operationen eines RBF-Netzes liegt in [2] vor.

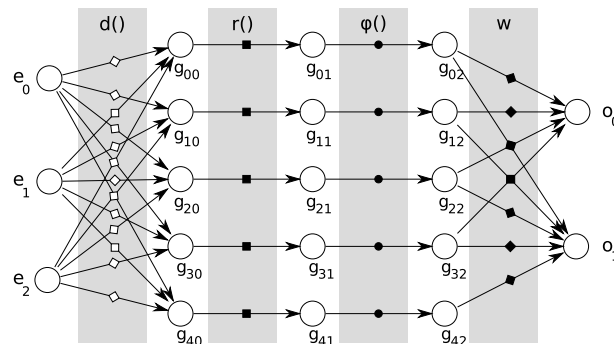


Abbildung 1. Durchflussgraph für ein RBF-Netz mit 3 Eingaben, 5 RBF-Neuronen und 2 Ausgabe-Neuronen

Der Einsatz von RBF-Netzen für die Fehlerdiagnose wurde von Leonard [4] untersucht. Klassische Multi-Layer Feed-Forward Netze, die durch Backpropagation trainiert werden, realisieren eine effektive Trennung der Trainings-Beispiele anhand der Klasse. Trennungs-Probleme treten aber dann auf, wenn zu wenige Beispiele zum Training verwendet wurden. Als Ergebnis ist die Trennung der Klassen nicht optimal, und eine Eingabe, die während der Trainings-Phase nicht gelernt wurde, wird nicht richtig klassifiziert. Weil RBF-Netze wie *lokale Approximatoren* funktionieren, sind diese in der Lage, nicht klassifizierbare Eingaben zu erkennen.

3 Ansätze zur Parallelisierung

RBF-Netze werden meistens durch Neuronen-Parallelismus parallelisiert. Während der Parallelisierung müssen zwei Schritte erfolgen. Die Verteilung der RBF-Neuronen auf alle Prozesse, ist ein erster Schritt, in der die Neuronen-Menge partitioniert wird. Der zweite Schritt

der Parallelisierungs-Methode berücksichtigt die Modellierung der Kommunikation, die zwischen den Prozessen stattfindet.

Es wurden unterschiedliche Methoden entwickelt, um die Ausgabe-Werte eines Netzes in einer parallelen Umgebung zu berechnen. Die *einfache Methode* folgt der theoretischen Funktionsweise der RBF-Netze. Eine Verbesserung dieser Methode wird in der Arbeit von Pomerleau [6] beschrieben. Die Methode wird als Methode der *partiellen Summen* bezeichnet und Pomerleau deutet darauf hin, dass die Gewichte Eigenschaften der RBF-Neuronen sind, was Auswirkungen auf das Kommunikations-Modell hat.

Im Laufe der parallelen Simulation eines RBF-Netzes müssen die Teilergebnisse, die von den einzelnen Prozessen berechnet wurden, gesammelt und verarbeitet werden. Die Prozesse müssen untereinander kommunizieren, um einen globalen konsistenten Zustand des Netzes zu erreichen, d.h. die gesamten Ausgabe-Werte müssen berechnet und gesammelt werden. Jeder Prozess sendet an alle anderen seine Teilergebnisse, was insgesamt für alle Prozesse T_{dist} Zeiteinheiten dauert, während die Zentralisierung der Ausgabe-Werte $T_{collectone}$ Zeiteinheiten dauert.

Im Laufe der parallelen Trainings-Phase ist es nötig, dass alle Prozesse die Fehlerabschätzung der Propagierung kennen, um die Gradienten der Neuronen-Parameter zu berechnen. Das bedeutet, dass die Ergebnisse der einzelnen Prozess-Propagierungen *irgendwo* gesammelt und verarbeitet werden müssen, um danach die Ergebnisse an alle Prozesse zurück zu verteilen. Sei $T_{collectall}$ der Zeitaufwand, die globale Netz-Fehler-Abschätzung an alle Prozesse zu verteilen. Diese Zeiten beeinflussen die Zeit einer Propagierung durch das Netz und wiederholen sich für alle Trainings-Daten einer Epoche.

Mit f_{comm} wird die Übertragungszeit einer Dateneinheit bezeichnet. Diese ist eine Funktion, deren Argumente nicht bekannt sind, weil diese vom Übertragungs-Medium, Kommunikations-Muster u.a. abhängig sind.

RBF-Neuronen-Parallelismus RBF-Neuronen-Parallelismus ist eine Art des Neuronen-Parallelismus, bei dem die RBF-Neuronen auf p Prozesse verteilt werden. Die Ausgabe-Neuronen werden aber auf alle Prozesse dupliziert. Das bedeutet, dass ein Prozess g/p RBF-Neuronen und n Ausgabe-Neuronen besitzt. Dieser Sachverhalt wird in Abbildung 2 beispielhaft dargestellt.

Sei ein RBF-Netz mit 6 RBF-Neuronen $g_i, i = 1..6$ und 2 Ausgabe-Neuronen $o_j, j = 0, 1$. Die Dimension der Eingaben für ein RBF-Neuron beträgt 3 ($e_k, k = 1..3$). Das RBF-Netz (Abbildung 2) wird als Durchfluss-Graph dargestellt. Zuerst werden die Entfernungen berechnet, danach folgen die Aktivierungen der RBF-Neuronen, und zuletzt werden die partiellen Summen berechnet. Die partiellen Summen werden in den Knoten $s_{lj}, l = 1, 2$ und $j = 1, 2$ kumuliert und danach an die Ausgabe-Neuronen $o_j, j = 0, 1$ gesendet.

Weil die Ausgabe-Neuronen dupliziert werden, müssen alle partiellen Summen Knoten s_{lj} ihre Werte p -Mal an die Ausgabe-Neuronen senden. In der Abbildung sendet s_{10} zweimal seinen Wert, einmal an o_0 aus Prozess-1 und einmal an o_0 aus Prozess-2. Die Anzahl der Knoten pro Prozess, in denen partiellen Summen kumuliert werden, beträgt n .

T_{dist}^g , $T_{collectone}^g$ und $T_{collectall}^g$ werden für diesen Parallelisierungs-Ansatz wie folgt ausgedrückt:

$$T_{dist}^g = p \log(p) f_{comm} n \quad (1)$$

$$T_{collectall}^g = 0 \quad (2)$$

$$T_{collectone}^g = 0 \quad (3)$$

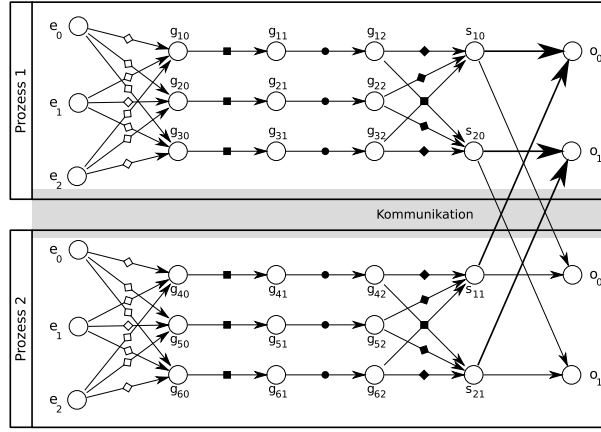


Abbildung 2. RBF-Neuronen Parallelismus

RBF- und Ausgabe-Neuronen-Parallelismus Bei diesem Ansatz enthält jeder Prozess g/p RBF-Neuronen und n/p Ausgabe-Neuronen (siehe Abbildung 3). Die Kommunikation erfolgt, indem die Prozesse die partiellen Summen an die anderen Prozesse senden und danach die berechneten Werten sammeln. Aus diesem Grund sind $T_{collectone}^o$ und $T_{collectall}^o$ nicht mehr Null. Für T_{dist}^o , $T_{collectone}^o$ und $T_{collectall}^o$ entstehen folgende Gleichungen, die den Kommunikationsaufwand abschätzen.

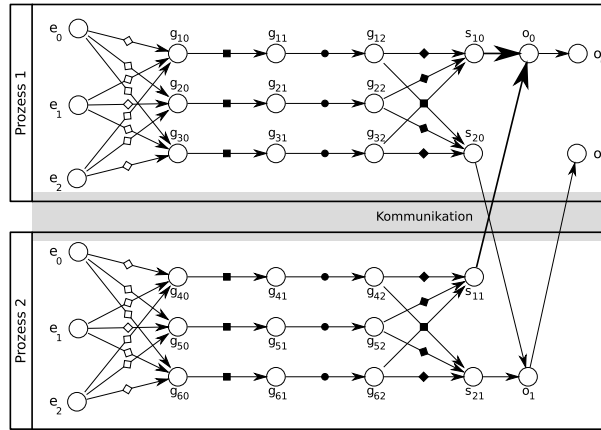


Abbildung 3. RBF- und Ausgabe-Neuronen Parallelismus

$$T_{dist}^o = p(p-1) \frac{n}{p} f_{comm} = n(p-1) f_{comm} \quad (4)$$

$$T_{collectall}^o = (p-1) f_{comm} \quad (5)$$

$$T_{collectone}^o = (p-1) \frac{n}{p} f_{comm} \quad (6)$$

Stufen-Parallelismus In diesem Szenario werden die Ausgabe-Neuronen einem einzigen Prozess zugeordnet, und die restlichen $p-1$ Prozesse enthalten je $g/(p-1)$ RBF-Neuronen. Es entstehen also zwei Stufen, die der RBF-Neuronen (aus $p-1$ Prozessen) und die der Ausgabe-Neuronen (aus 1 Prozess). Die Arbeitsweise eines parallelen Netzes mit Stufen-Parallelismus ist in Abbildung 4 dargestellt.

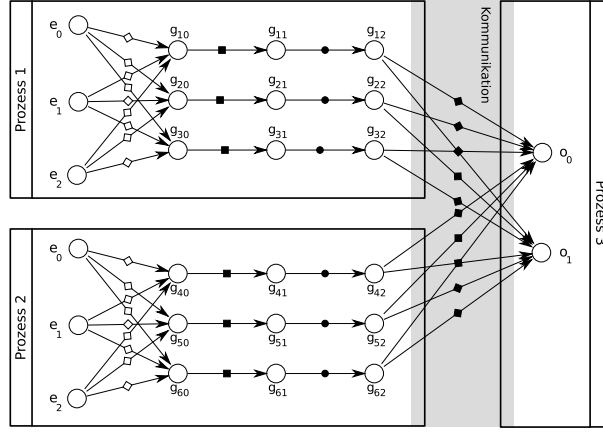


Abbildung 4. Stufen-Parallelismus

Die Partitionierung in dieser Situation wird nicht mehr durch die Methode von Pomerleau realisiert, und der Kommunikationsaufwand ist deswegen von g abhängig. Als Folge ändern sich die Ausdrücke für T_{dist}^s , $T_{collectone}^s$ und $T_{collectall}^s$ entsprechend:

$$T_{dist}^s = (p-1) \frac{g}{p-1} f_{comm} = g f_{comm} \quad (7)$$

$$T_{collectall}^s = n \log(p-1) f_{comm} \quad (8)$$

$$T_{collectone}^s = 0 \quad (9)$$

4 Umsetzung der Parallelisierung

RBF++ [3] ist eine Implementierung eines Simulators für RBF-Netze, der an der Technischen Universität München realisiert wurde. Für das Training der Netze verwendet RBF++ den RPROP-Algorithmus [7]. Der Quellcode des Simulators ist für nicht kommerzielle Zwecke offen und enthält keine parallele Funktionalitäten. Die vorher vorgestellten Parallelisierungs-Ansätze wurden nach der Vorbereitung des Quellcodes von RBF++ praktisch umgesetzt.

Die Parallelisierungs-Ansätze sind *conceptual models* [8]. Solche Modelle beschreiben Prozess-Abläufe, doch enthalten sie keine Hinweise auf die Modellierung der *praktischen* Anwendung. Aus einer *Model Driven Architecture* (MDA) Perspektive kann die Trennung zwischen Modell und Technik realisiert werden. Das *Platform Independent Model* (PIM) betrachtet ausschließlich die rein fachlichen Aspekte und die Modellierung eines Designs. Das Ergebnis der Umsetzung aus der Modell-Ebene in die Technik-Ebene erfolgt durch eine Modelltransformation, d.h. es findet eine Transformation aus einem PIM in ein PSM (*Platform Specific Model*) statt.

Eine Folge dieser Arbeitsweise ist die Möglichkeit, die Kommunikations-Muster für die *praktische Welt* zu übersetzen. Es wurde eine Nachrichten-Kommunikation mit MPI gewählt. MPI stellt ein Nachrichtenaustausch-Paradigma dar, weil in einer verteilten/parallelen Umgebung die Kommunikation auch über Nachrichten erfolgen kann. MPI ist eine standardisierte Schnittstelle, die Namen, Methoden und deren Ausgaben bestimmt. Die Übersetzung der Kommunikations-Muster in MPI-Operationen erfolgt durch einen *Schlüssel*, die Art des Kommunikations-Musters, und erlaubt dadurch die Transformation aus dem PIM in ein PSM zu realisieren.

5 Ergebnisse

Allgemein soll die Parallelisierung eines RBF-Netzes nicht die Funktion des Netzes beeinträchtigen. Deshalb kann, um den Speed-Up der parallelen Version vorauszusehen, die sequentielle Version eines Netzes mit der parallelen Version des Netzes verglichen werden.

RBF-Netze sind wegen ihrer Architektur hoch-parallelisierbar, doch innerhalb der Trainings- und der Simulations-Phase existieren einige Kommunikations-Stellen, die im Laufe der sequentiellen Version nicht auftreten. Die Kommunikation in einer parallelen RBF-Netz-Umgebung wird benötigt, um einen einheitlichen Zustand des Netzes über alle p Prozesse aufrecht zu erhalten. S_t bezieht sich auf dem zeitlichen Speed-Up, d.h. wieviel schneller *funktioniert* ein paralleles Netz (T_p) im Vergleich mit einem sequentiellen Netz (T_s) (Formel 10).

$$S_t = \frac{T_s}{T_p} \quad (10)$$

Die Parallelisierungs-Ansätze besitzen spezifische Eigenschaften, die durch verschiedene Messverfahren untersucht wurden. Dafür wurden mehrere Experimente entworfen und realisiert. Während der Simulation müssen Eingabe-Daten so schnell wie möglich klassifiziert werden. Im Laufe des Trainings müssen die Netze schnell trainiert werden, auch wenn das Training eine untergeordnete Rolle für diese Arbeit spielt. Aus diesen Gründen werden die *Performance-Messung*, die *Qualitäts-Messung* und *Validierungs-Messung* vorgestellt.

Performance-Messungen. Die Geschwindigkeit des Trainings und der Simulation ist von den Test-Daten nicht abhängig, daher werden für die *Performance-Messung* künstliche Daten verwendet. Während der Ausführung sind die Eingabedaten-Vektoren in jeder Komponente mit 0.5 initialisiert, die Gewichte der synaptischen Verbindungen werden mit 0.5 initialisiert, die RBF-Neuron-Breiten sind 0.7 und die Ausgabe-Neuron-Schwellwerte sind 0.1. Die RBF-Netz-Topologie wird ständig geändert und die Messungen werden für verschiedenen Anzahlen von Prozessoren durchgeführt. In Tabelle 1 werden die Eigenschaften des Messverfahrens für die Simulation dargestellt. Für jede Konfiguration werden die Messungen drei Mal wiederholt, aber während der Messungen wird nur die RBF-Neuronen-Anzahl berücksichtigt. Das bedeutet, dass für eine bestimmte RBF-Neuronen-Anzahl 21 Messungen realisiert werden, die danach gemittelt werden.

Parameter	Simulation		Training	
	Wertebereich Inkrement		Wertebereich Inkrement	
RBF-Neuronen-Anzahl	100 bis 1000	100	300 bis ..	1
Ausgabe-Neuronen-Anzahl	50 bis 200	25	10	0
Prozess-Anzahl	2 bis 16	2	1, 2, 4, 8 und 16	-
Wiederholung der Messung	3	0	1	0

Tabelle 1. Performance-Messparameter

In Abbildung 5 werden die erreichten Speed-Ups der Parallelisierungs-Ansätze graphisch verglichen. Die Speed-Ups wurden durch die Formel 10 berechnet und beziehen sich auf die nicht parallelisierte Simulation des Netzes. Es ist zu beobachten, dass RBF-Neuronen-Parallelismus und RBF- und Ausgabe-Neuronen-Parallelismus fast gleiche Speed-Ups erreichen, wobei der Speed-Up des Stufen-Parallelismus geringer ausfällt.

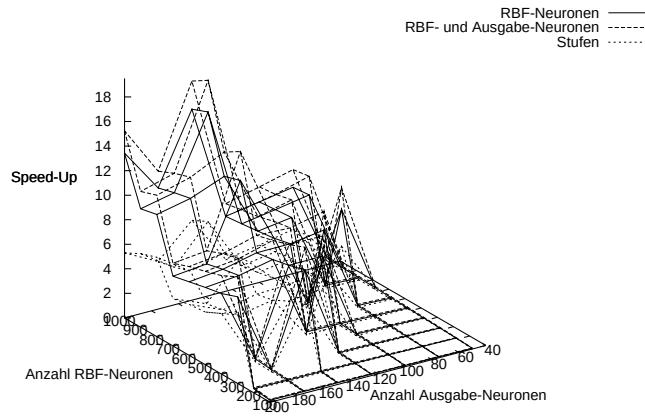


Abbildung 5. Vergleich der erreichten Speed-Ups für 16 Prozesse

Qualitäts-Messungen. Nach der Trainings-Phase eines parallelen RBF-Netzes sollen die Trainings-Ergebnisse eine bestimmte Güte-Charakteristik erreichen. Die Erfüllung dieses Kriteriums wird durch das Qualitäts-Messverfahren beurteilt (siehe Tabelle 2).

Fehlerdiagnose ist eine Klassifizierungs-Aufgabe, und weil es schwer ist, Test-Daten für die Fehlerdiagnose zu erzeugen [5], wurden analytische Daten aus dem UCI Machine Learning Repository [9] verwendet. Der ausgewählte Datensatz enthält 28×28 Pixel große Grauwert-Bilder der Ziffern von 0 bis 9. Für jeder Ziffer ist die Anzahl der Bilder unterschiedlich, insgesamt sind es 958 Bilder.

Die Informationen aus den Bildern wurden als Eingabe-Daten der RBF-Netze verwendet. Daher wurden, um die Fehlerdiagnose zu *simulieren*, die Bilder verarbeitet:

- Skalierung: die Dimension wurde auf 10×10 Pixel verändert;
- Transformation der Werte: die Grauwerte $[0, 255]$ wurden auf das Intervall $[0, 1]$ transformiert.

Der analytische Test-Datensatz enthält also 958 Test-Daten der Dimension 100, die in 10 Klassen klassifiziert werden können. Es folgt, dass ein RBF-Netz, das mit diesen Daten arbeiten soll, RBF-Neuronen, die Eingabe-Daten der Dimension 100 akzeptieren, und 10 Ausgabe-Neuronen enthalten muss.

Parameter	Qualität	
	Wertebereich	Inkrement
RBF-Neuronen	100 bis 900	1
Ausgabe-Neuronen	10	0
Prozessen-Anzahl	2 bis 16	2
Wiederholung der Messung	6	0

Tabelle 2. Qualitäts-Messparameter

In Abbildung 6 sind die Mittelwerte des Propagierungs-Fehlers für jede Epoche einer Trainings-Phase dargestellt. Die erste Folge des Trainings mit analytischen Daten ist, dass die Propagierungs-Fehler unterschiedliche Entwicklungen nehmen. Die Graphiken für 2, 6 und 14 Prozesse sind nicht ähnlich. Es kann beobachtet werden, dass für 6 Prozesse die Fehler von Anfang an des Trainings hohe Werte besaßen (größer als 0.5), und dass diese

danach gesunken sind, weil der Trainings-Algorithmus die RBF-Neuronen löscht, die einen hohen Beitrag an die Fehler-Bildung haben. Eine andere Beobachtung ist, dass die 3 RBF-Netze unterschiedlich viele Epochen trainiert wurden. Für 14 Prozesse wurden weniger als 1500 Epochen benötigt, im Gegensatz zu mehr als 2000 für 6 Prozesse.

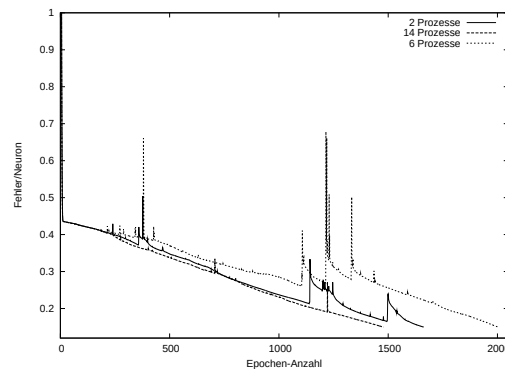


Abbildung 6. Entwicklung des Propagierungs-Fehlers

Validierungs-Messungen. Die mathematischen Modelle der Parallelisierungen sollen durch ein Messverfahren in der Praxis validiert werden, und dafür wurde die *Validierungs-Messung* realisiert (siehe Tabelle 3). Die Ergebnisse dieses Messverfahrens werden als *Beweis aus der Praxis* ausgewertet und werden die mathematischen Formeln entweder bestätigen oder entkräften.

Parameter	Validierung	
	Wertebereich	Inkrement
RBF-Neuronen-Anzahl	400 bis 2000	100
Ausgabe-Neuronen-Anzahl	100 bis 1000	100
Prozess-Anzahl	2 bis 16	2
Wiederholung der Messung	7	0

Tabelle 3. Validierungs-Messparameter

Die Messungen wurden an die mathematischen Modelle angeglichen (durch eine Fit-Methode¹), um den f_{comm} Parameter zu berechnen. Dieser ist ein Maß des Kommunikationsaufwandes für einer Dateneinheit, die zwischen den Prozessen gesendet wird. Die Ergebnisse sind in Tabelle 4 aufgelistet.

Parallelismus	Formel Kommunikation	Standard	
		(f_{comm} in μs)	Abweichung
RBF-Neuronen	$np \log(p-1)f_{comm}$	6.46	3.67%
RBF- und Ausgabe-Neuronen	$n(p-1)(1+1/p)f_{comm}$	8.11	4.97%
Stufen	gf_{comm}	13.30	3.67%

Tabelle 4. Werte des Kommunikationsaufwandes nach dem Ausgleich mit den Messungen

¹ Gnuplot verwendet die Methode der kleinsten Quadrate für den Ausgleich unbekannter Parameter

Die Kommunikation während der parallelen Simulation eines RBF-Netzes besitzt unterschiedliche Eigenschaften, je nachdem, welcher Parallelisierungs-Ansatz angewendet wird. Folgende Schlussfolgerungen können nach dem Validierungs-Test gezogen werden:

- Der Kommunikationsaufwand kann durch mathematische Modelle ausgedrückt werden.
- Der Kommunikationsaufwand/Dateneinheit ist für den Stufen- Parallelismus am höchsten ($f_{comm} = 13.30 \mu s$).
- Mit dem RBF-Neuronen-Parallelismus wird ein kleiner Quotient Kommunikationsaufwand per Dateneinheit erzeugt ($f_{comm} = 6.46 \mu s$), aber aufgrund der vielen Übertragungen, die benötigt werden, entstehen für viele Ausgabe-Neuronen hohe Kommunikations-Zeiten (maximal 1.2 s für 1000 Ausgabe-Neuronen).

Der hohe Mittelwert für den Stufen-Parallelismus entsteht weil die Datenpakete, die ein Prozess sendet, groß sind, und der Kommunikationsaufwand mit der Anzahl von RBF-Neuronen linear steigt. Für 300 RBF-Neuronen sind die Kommunikations-Zeitdauern kleiner als für die anderen Parallelisierungs-Ansätze und für 1000 RBF-Neuronen sind die Zeitdauern viel höher. Aus diesen Gründen ergibt sich ein so hoher Mittelwert.

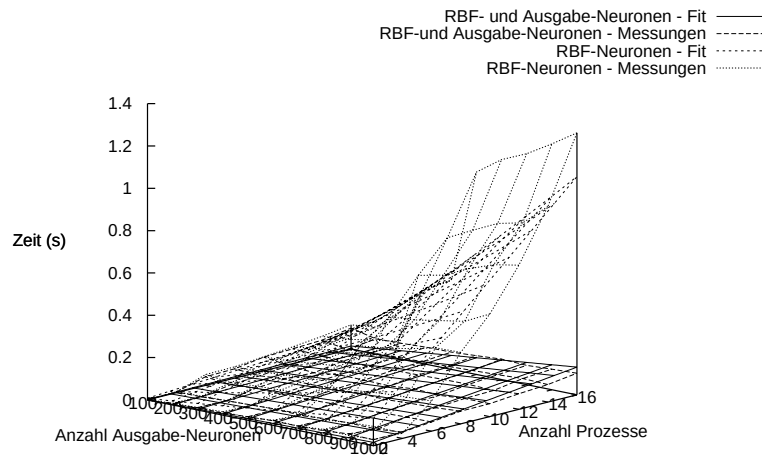


Abbildung 7. Ausgleich der mathematischen Modelle mit den Messungen für RBF-Neuronen-Parallelismus und RBF- und Ausgabe-Neuronen-Parallelismus

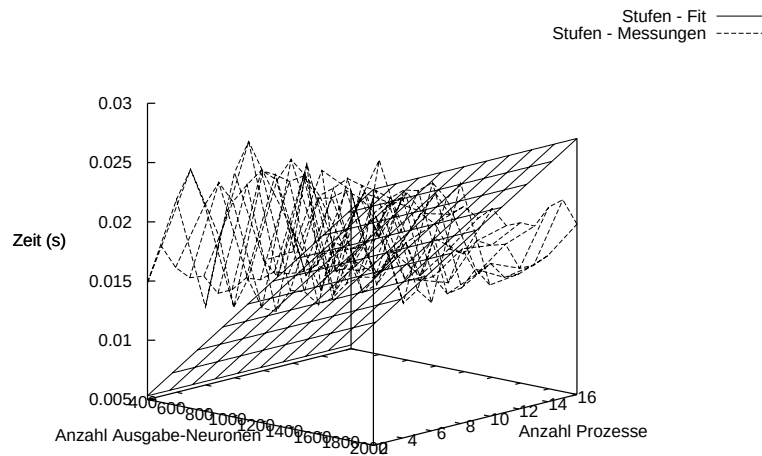


Abbildung 8. Ausgleich des mathematischen Modells mit den Messungen für Stufen-Parallelismus

6 Zusammenfassung

Die im Rahmen dieser Arbeit vorgestellten Ansätze zur Parallelisierung eines Diagnose-Systems wurden durch die aus den Messungen erzeugten Messdaten ausgewertet. Die entscheidende Rolle für das Erreichen von hohen Speed-Ups spielt die Kommunikation, die zwischen den Prozessen einer parallelen Umgebung notwendig wird. Die Last, die von den Berechnungen ausgeht, die in einem RBF-Netz stattfinden, beeinflusst ebenfalls die erreichten Speed-Ups. Ein *Gleichgewicht* zwischen dem Kommunikationsaufwand und der Last der Berechnungen ist der Schlüssel hoher Geschwindigkeiten der Fehlerdiagnose in einem Online-Fehlerdiagnose-System.

Durch RBF-Neuronen-Parallelismus wird in der Theorie ein hoher Kommunikationsaufwand erzeugt, der auch in der Praxis bestätigt wurde. Dieser Ansatz ist für RBF-Netze mit wenigen Ausgabe-Neuronen geeignet, weil die Kommunikation den gleichen Aufwand wie der des RBF- und Ausgabe-Neuronen-Parallelismus generiert.

RBF- und Ausgabe-Neuronen-Parallelismus zeigen sich als ein geeigneter Ansatz, RBF-Netze mit vielen RBF- und Ausgabe-Neuronen zu parallelisieren. Dafür verlangt dieser Parallelismus aber einen stärker ausgearbeiteten Kommunikations-Mechanismus, weil die Verteilung der Ausgabe-Neuronen die Komplexität der Implementation erhöht.

Stufen-Parallelismus ist ein interessanter Ansatz, der aber in der Arbeit vorgestellten Form für RBF-Netze nicht angemessen ist, weil die Ausgabe-Stufe, die alle Ausgabe-Neuronen enthält, langsamer als die RBF-Neuronen-Stufe ist. Der Kommunikationsaufwand, der von dem Stufen-Parallelismus erzeugt wird, ist hoch, stellt aber nicht die Schwäche des Ansatzes zur Parallelisierung dar.

Der Einsatz von parallelen RBF-Netzen für die Fehlerdiagnose in großen Automatisierungs-Anlagen soll weiter untersucht werden. Die Erzeugung von RBF-Netzen mit guten Diagnose-Leistungen und deren Einsatz für umfangreiche Diagnose-Probleme ist ein wichtiges Ziel zukünftiger Forschungen. Zusätzliche Erweiterungen dieser Arbeit beziehen sich auf eine genauere Betrachtung und Modellierung der Kommunikation innerhalb paralleler Diagnose-Systemen, um diese optimal durch Parallelisierung zu beschleunigen.

Literatur

1. Markou, M., Singh, S.: Novelty detection: a review-part 2: neural network based approaches. In: Signal Processing, Vol. 83, Nr. 12., Elsevier (2003) 2499–2521
2. Haykin, S.: Neural Networks: A Comprehensive Foundation. Prentice Hall PTR Upper Saddle River, NJ, USA (1998)
3. Buck, S.: RBF++ A C++ Library for Function Approximation Using Networks of Radial Basis Functions. (1999)
4. Leonard, J.A., Kramer, M.A.: Radial basis function networks for classifying process faults. In: Control Systems Magazine, Vol. 11, Nr. 3. IEEE Computer Society (1991) 31–38
5. Isermann, R.: Fault-Diagnosis Systems: An Introduction From Fault Detection To Fault Tolerance. Springer-Verlag, Berlin Heidelberg New York (2006)
6. Pomerleau, D.A., Gusciora, G.L., Touretzky, D.S. and Kung, H.T.: Neural network simulation at Warp speed: how we got 17 million connections per second. In: Neural Networks, 1988, IEEE International Conference on. IEEE Computer Society (1988), 143–150
7. Riedmiller, M. and Braun, H.: A direct adaptive method for faster backpropagation learning: theRPROP algorithm. In: Neural Networks, 1993., IEEE International Conference on. IEEE Computer Society (1993) 586–591
8. Daniels, J.: Modeling with a sense of purpose. In: IEEE software, Vol. 19, Nr. 1. IEEE Computer Society (2002) 8–10
9. A. Asuncion, D.J. Newman: UCI Machine Learning Repository. <http://www.ics.uci.edu/~mllearn/MLRepository.html>. University of California, Irvine, School of Information and Computer Sciences (2007)

GPU accelerated scientific computing: Fluid and particulate flows with CUDA

Tobias Hahn, Vincent Heuveline, Björn Rucker

Karlsruhe Institute of Technology (KIT), University of Karlsruhe (TH)
Department of Mathematics
Institute for Applied and Numerical Mathematics
AG Numerical Simulation, Optimization and High Performance Computing,
76128 Karlsruhe, Germany
`tobias.hahn@kit.edu`, `vincent.heuveline@kit.edu`,
`bjoern.rucker@kit.edu`

Abstract. Simulations of particulate flows, which involve gases and liquids with suspended solid particles like dust, are generally highly CPU-time demanding. The question arises whether such computations can be performed on the GPU applying highly parallel programming models like CUDA. In this paper we demonstrate that numerical simulation in that context can greatly benefit from these emerging technologies and present results in a 2D and 3D setup. We evaluate the NVIDIA GT200 GPU by benchmarking basic scientific kernels, a ported 2D fluid flow simulation and a 3D particulate flow simulation, based on a model particularly fitted to the GPU architecture.

1 Introduction

Various applications related to fluid flow problems still represent a challenging task with respect to numerical simulation. In many cases even with high-speed supercomputers only rough approximations of the solution can be achieved in reasonable time. The computation of particulate flows, which describe gases and liquids with suspended solid particle like coal dust, is typically highly computation-time demanding.

In this paper we propose a GPU-based approach for the numerical simulation of the associated problems utilizing CUDA as a programming model. In that context we present the hardware architecture of the NVIDIA GT200 GPU and give performance evaluations both for the overall simulation and the basic scientific kernels involved. Among the benchmarked kernels we consider operations on vectors, sparse matrices as well as fast Fourier transforms.

2 Advances in GPGPU and the CUDA programming model

General purpose computation on graphics processing units (GPGPU) describes calculations done on a GPU, whose primary purpose is not video output. The problems GPGPU deals with were originally strongly connected to computer graphic and image processing. In general all applications that run efficiently on a GPU show a high rate of parallelism within the underlying model or within the algorithm to solve the model equations.

The introduction of full floating-point support on GPUs motivated lots of scientific projects to implement e.g. finite difference and finite element techniques for the solution of systems of partial differential equations (PDEs) using GPU hardware. In 2003 several papers demonstrated e.g. solutions of the Navier-Stokes equations for incompressible fluid flow on the GPU [4,5] or other boundary value problems [6].

With the introduction of built-in double-precision support and partial IEEE 754 compatibility, GPUs evolve towards universally usable processing units. Still, their paradigm is related to former graphics stream processing: The same series of operations is applied to every element of a set of data (stream). Operations of a kernel are pipelined, such that many stream processors can process the stream in parallel. The main problem in this context is

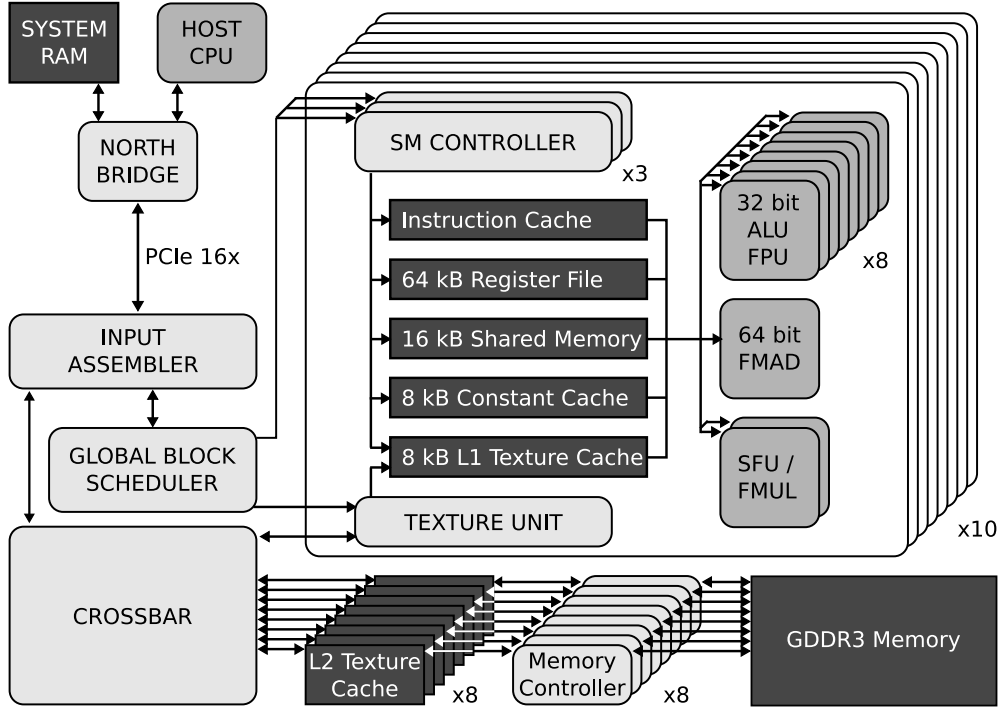


Fig. 1. Simplified architecture of the of the NVIDIA GT200 GPU

memory latency, especially when data dependency is high and data locality is low. GPUs always try to hide memory latency by executing many kernel instances in parallel on the same core. Switching these lightweight “threads” and operating on another register set can be done in just a few cycles whereas fetching data from the global memory costs several hundreds of cycles.

While the above problem is inherent for many-core computing, other restrictions of stream processing techniques have been addressed in NVIDIA’s Compute Unified Device Architecture (CUDA). CUDA offers e.g. gather and scatter operations on the global graphics memory. Furthermore CUDA devices can be programmed with slightly extended C and runtime libraries, including hardware support for double precision (obeying IEEE 754).

Figure 1 outlines the hardware architecture of the NVIDIA GT200 GPU: The processing element consists of 10 clusters, each containing three streaming multiprocessors (SM). Those again incorporate eight single precision FPUs, one double precision unit, two super function units and several caches, the most important being a fast 16 kB shared memory facilitating the interaction of the FPUs. While the SM executes one thread per SP in a SIMD fashion, additional threads wait for activation in further register sets. In the case of divergence, threads are processed in sequence as usual. Different SMs may be at different points within the kernel, such that the whole chip works in a MSIMD (multiple SIMD) way. Further details of the chip are presented in [1].

In order to manage these $8 \times 3 \times 10 = 240$ processing units, threads are structured into blocks and grids. Each SM works on one thread block at a time, while the SM controller decides in which order to execute the up to 512 threads per block. Grids are issued one after another on the whole GPU, meaning that data exchange is not possible for threads of different grids.

For a thread to locate its position within a block and grid, built-in variables are available returning the respective index. In general, non-consecutive memory accesses of those many processors cannot be satisfied without delay, even with highly banked memory. Details on performance losses and recommendations to avoid them are covered in [2,11].

Other extensions to the C programming language are just function and variable qualifiers determining the target location and a construct to launch kernels on the GPU. By utilizing a pre-compiler, device and host code are separately compiled and linked together in a later step. This procedure is simplified by the NVIDIA compiler driver *nvcc*.

3 Description of models and challenges within the solution process

In this paper we only focus on direct numerical simulation (DNS) to solve the governing equations (1). In order to resolve all scales relevant for solving the problem, DNS has high requirements on computational power and memory. This makes it necessary to not only focus on the numerical scheme but on the hardware platform as well. Increasing the number of conventional processors is an effective but not economic solution. Accelerators such as the GT200 GPU promise multiple computational performance at low financial effort and moderate power consumption compared to present CPUs. In the following we are investigating under what conditions GPUs can be used to accelerate the computation.

The governing equations for both application problems discussed in this paper are the time-dependent incompressible Navier-Stokes equations arising from continuum mechanical descriptions of fluid behavior:

$$\rho \frac{D\mathbf{u}}{Dt} = -\nabla p + \mu \Delta \mathbf{u} + \mathbf{f}, \quad \nabla \cdot \mathbf{u} = 0, \quad (1)$$

where $\mathbf{u}$ is the fluid velocity field, p the pressure field, ρ the constant fluid density, μ its molecular viscosity constant and $\mathbf{f}$ combines external forces acting on the fluid. The operator D depicts the non-linear material derivative.

Both, the meteorological and the particulate flow simulation are based on these equations, completed by initial and boundary conditions. In the case of particulate flows, we add equations describing the particle behavior and the interaction with the fluid, what will be discussed in section 3.2.

3.1 A meteorological model

An existing meteorological implementation for simulating tropical cyclones in 2D was considered for acceleration with CUDA. The task is to solve the emerging Navier-Stokes equations (1) on a square geometry, as exemplary shown in Figure 2. The implemented algorithm accomplishes this on the Navier-Stokes equations in their stream vorticity formulation which is frequently used in 2D simulations and can be found in e.g. [12,13].

A transformation of the fluid equations from the primal variable (u, v, p) to a vorticity ξ based point of view is performed and the stream function Ψ is used to keep the velocity information for u and v , where $\mathbf{u} = (u, v)^T$. By using the abbreviation $\partial_i X = \frac{\partial X}{\partial i}$ the functions Ψ and ξ can be written as follows:

$$\xi = \partial_x v - \partial_y u \quad \text{and} \quad u = -\partial_y \Psi, v = \partial_x \Psi. \quad (2)$$

By applying the curl operator on equation (1) the pressure term vanishes and with ξ and Ψ as above and $\mathbf{f} = (f_1, f_2)^T$ it remains:

$$-\Delta \Psi = \xi, \quad (3)$$

$$\partial_t \xi + u \cdot \partial_x \xi + v \cdot \partial_y \xi - \mu \Delta \xi = \partial_x f_2 - \partial_y f_1. \quad (4)$$

The discretization for solving the Poisson equation (3) for a fixed initial ξ is done by finite differences and the arising linear system is solved by an FFT. With this solution for ξ

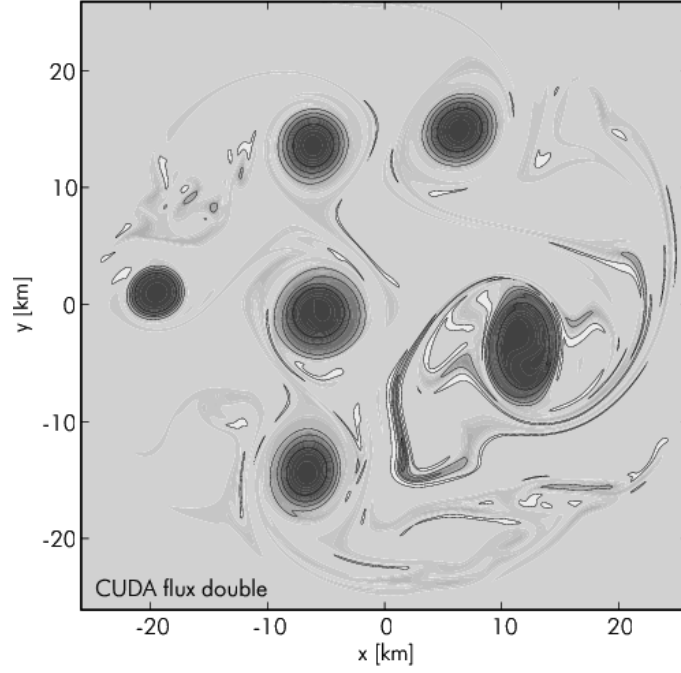


Fig. 2. Vorticity plot produced by the 2D meteorological application for simulating tropical cyclones. On a square domain the incompressible Navier-Stokes equations are solved in their stream vorticity formulation.

it is possible to compute u and v via (2) and update ξ by using equation (4). This procedure is then continued iteratively. The proof for the equivalence of the Navier-Stokes equations in their stream vorticity formulation to the primal variable based formulation with periodic boundary conditions can be found in [15].

Numerical analysis and experiments have shown, that the algorithm produces reasonable results also with single precision FFT, a necessity, as NVIDIA's CUFFT library is currently not available in double precision. The CPU implementation is based on Intel MKL's DFTI routines and is very well optimized for the used Intel-based hardware.

3.2 A model for particulate flows

From the numerical methods developed for simulating particulate flows, the most popular are the Arbitrary Lagrangian Euler (ALE) techniques, as pursued in [7,8] and the Fictitious domain method [9,10]. The ALE technique allows high resolution on the particle surface by remeshing the fluid domain every few time steps. This is computationally very expensive and generates an unstructured grid which in turn usually leads to irregular data access patterns that is not well suited for the GPU. Fictitious domain methods pretend the fluid to occupy the whole domain with particles and solve a pure fluid problem on this extended domain. The underlying discretization can be chosen fixed and structured, what makes this methods especially well suited for GPU implementation. In a second step the coupling is enforced, resulting in a corrected fluid velocity field complying with rigid body motion in the particle domain.

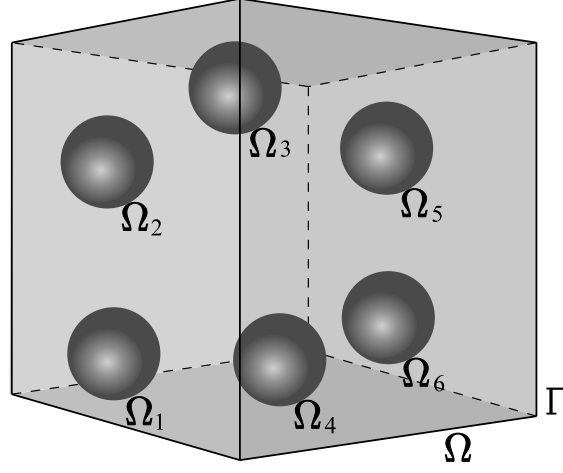


Fig. 3. Geometry of the 3D particulate flow setup. Ω denotes the complete area that is simulated, Ω_i denote the particles in the fluid and Γ is the boundary of the geometry.

Following [10] and using the notations as shown in Figure 3, we can describe the problem with the following formulae:

$$\rho \frac{D\mathbf{u}}{Dt} = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{F}, \quad \nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega, \quad (5)$$

$$\frac{d\mathbf{U}_i}{dt} = \mathbf{g} + \frac{\mathbf{c}}{\rho_i - \rho} - \frac{1}{(\rho_i - \rho)V_i} \int_{\Omega_i} \mathbf{F} d\mathbf{x} \quad \text{in } \Omega_i, \quad (6)$$

$$\boldsymbol{\omega}_i = \frac{1}{2V_i} \int_{\Omega_i} \nabla \times \mathbf{u} d\mathbf{x} \quad i = 1, \dots, n, \quad (7)$$

$$\mathbf{u} = \mathbf{U}_i + \boldsymbol{\omega}_i \times (\mathbf{x} - \mathbf{X}_i) \quad \text{in } \Omega_i. \quad (8)$$

The newly introduced variables $\mathbf{U}_i, \boldsymbol{\omega}_i, \mathbf{X}_i, V_i$ and ρ_i depict the i th particle's translational and angular velocity, its centroidal position, volume and density. $\mathbf{g}$ depicts gravity and $\mathbf{c}$ are collision forces between particles or with the boundary Γ . $\mathbf{F}$ is the fluid-structure interaction force, that is set zero in the first solution step and opposite to the hydrodynamic force during the correction. Note that equation (6) is basically Newton's second law, rewritten in the form $a = \frac{f}{m}$. The actual coupling is given in the last equation that enforces rigid body motion in the particle domain and implicitly no-slip conditions on the particles' surfaces.

In [11] it is proposed to use a finite element method based on a rectangular grid for simple geometries as in Figure 3. Each cube is then subdivided into six tetrahedra each, in order to compute the integrals over the particle domains more accurately without increasing the number of degrees of freedom.

All implementations of fictitious domain methods considered in [9,10,11] use operator splitting techniques to decouple the operators involved. While [10] treats advection with the method of characteristics, [11] uses a wave-like equation method to linearize the problem and allow conventional LES solvers to be applied.

These solvers depend mostly on matrix-vector multiplications (mv kernel) and vector-vector operations such as $a \cdot \mathbf{x} + \mathbf{y}$ ($axpy$ kernel) and $\mathbf{x} \cdot \mathbf{y}$ (dot product). The matrices involved are kept sparse in order to facilitate the solution process. Usually the data structure chosen is the compressed sparse row storage (CSR).

4 Performance evaluation

In order to better analyze the solver speed for the overall problem we have to investigate the performance of the hardware platform for the elementary kernels *csmv*, *axpy*, *dot* and *fft*. As we are dealing with an off-chip accelerator, we have to examine the bandwidth between host and GPU as well.

The results shown in the following Figures 4 to 8 have been obtained with ASUS EN-GTX280 graphics cards on an ASUS P5B deluxe mainboard with Intel P965 north bridge and an Intel Core 2 Duo E6600 CPU. The tests with BLAS and FFT routines were executed with the libraries MKL 10.0.1.014 and CUBLAS/CUFFT 1.0 using CUDA driver 177.13.

4.1 Elementary kernels

The north-bridge offers one PCIe slot connected with 16 lanes and another one with 4 lanes only. As plotted in Figure 4 we can achieve approximately 2.5 GB/s on the faster slot and 0.65 GB/s on the other one. This result clearly indicates that hybrid (CPU/GPU) or multi-GPU programs should communicate as seldom as possible, as the bandwidth is not sufficient for frequent transfers of larger data packages. Even these values are only achieved for the faster slot if non-pageable “pinned” memory is used. On the GPU, the peak bandwidth was measured to be approximately 115 GB/s, but as mentioned before all 240 SPs have to share this bandwidth.

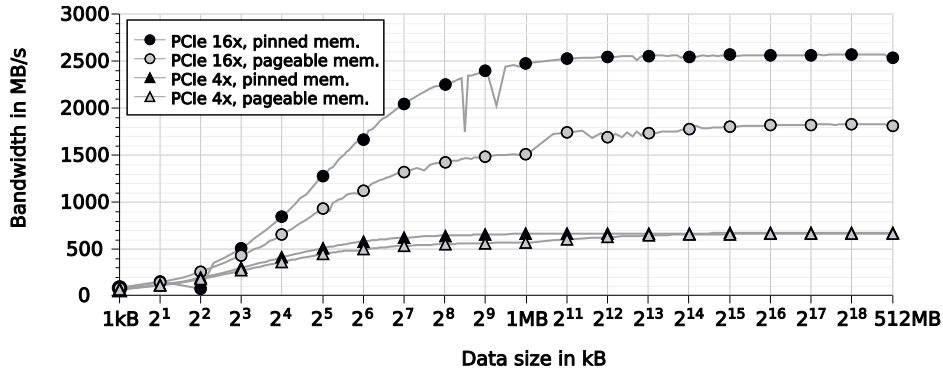


Fig. 4. Host-to-device memory bandwidth test results for different package sizes of the data, numbers of PCIe 1.1 lanes and modes of memory allocation.

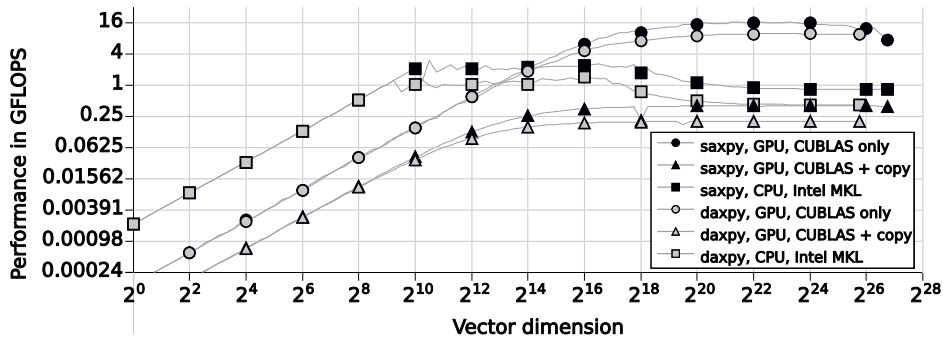


Fig. 5. Performance test results of the *axpy* kernel in single and double precision.

For the vector operation *saxpy* (single precision) and *daxpy* (double precision) the GPU can clearly outperform the CPU for large vector dimension (Figure 5). Due to a lack of

computational intensity and data re-use for this operation low performance is obtained with CUDA for small vectors.

The same is valid for the *dot* product test results, that resemble the ones of *saxy* very closely. Working exclusively on the GPU with large vectors is the key to achieving high speed-ups over MKL on a conventional CPU.

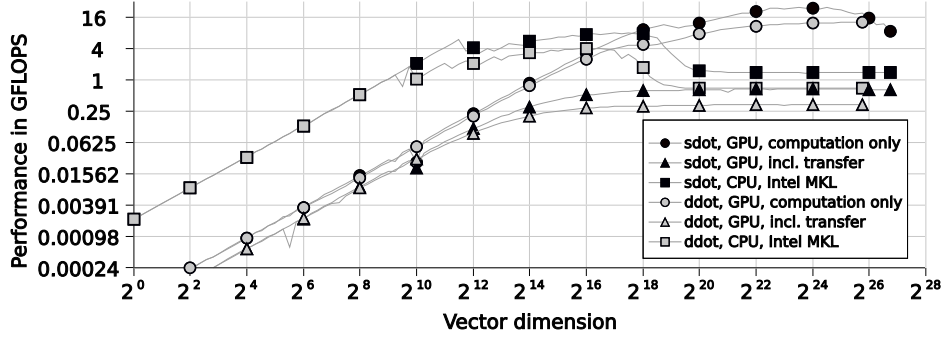


Fig. 6. Performance test results of the *dot* kernel in single and double precision

For the *csrmmv* test, $n \log n$ randomly placed entries of a matrix of dimension $n \times n$ were set non-zero. This is the worst-case for the GPU, as accesses to the source vector are random with high latency. Again a small number of entries is not suitable for the many cores of the GPU, but at a reasonable dimension of 8192, the speed-up compared to MKL increases to 2.4 (Figure 7).

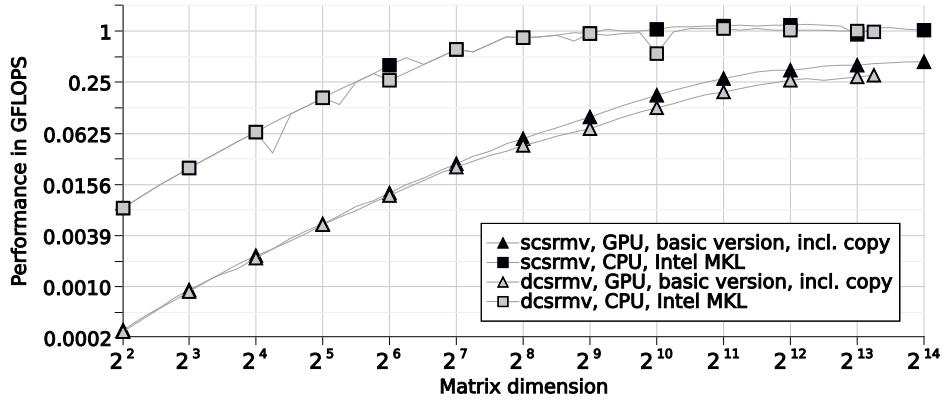


Fig. 7. Performance test results of the *csrmmv* kernel in single and double precision.

The measurements of the 2D FFT are done by forward and backward real transforms. Although the plots in Figure 8 show only speed-ups of 1.2 and 1.3, the complete meteorological problem solver demonstrates that in combination with other kernels, speed-ups increase significantly.

4.2 Applications in computational fluid dynamics

The use of GPUs in CFD went mostly hand in hand with Lattice Boltzmann methods. The choice for a LBM is obvious, as it operates on a finite difference grid, requires only directly neighboring data and can be implemented with stencil operations. The following performance results of the methods described above verify that also other approaches are well suited for GPU implementation.

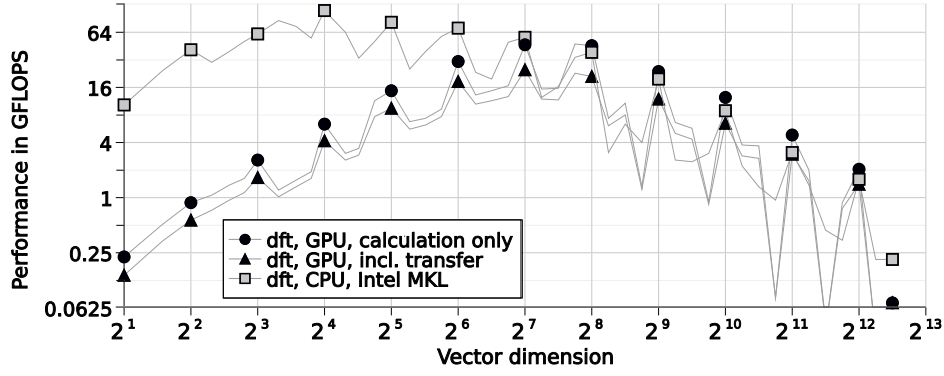


Fig. 8. Performance test results of the 2D real-to-complex *fft* in single precision.

2D fluid flow simulation: The first step in accelerating the existing simulation software was to create a Fortran interface for the CUFFT library and replace the MKL *fft*-calls. As expected from the *fft*-only benchmarks in Figure 8, we could just achieve minor improvements. It is to be underlined, that all tests could solely be done with CUFFT 1.0. Future versions are expected to achieve a higher performance.

The other part of the algorithm, numerical ODE solving via the Runge-Kutta method, was implemented with a ghost layer for the periodic boundary conditions in the Fortran version. When porting to the GPU, this layer had to be eliminated in order to keep a power-of-two element number per row and not introduce a moving offset, that would not suite the GPU's memory alignment constraints and lead to higher latencies.

As in the Fortran version, the integration was implemented via an out-of-place stencil operation. This allowed to declare the source data as textures, that are read-only but cachable (cf. Figure 1). In a last step of optimization, the needed grid points for the operation are explicitly prefetched into the GPU's shared memories. This leads to another performance improvement with final speed-ups of 3.5 to 4.5 depending on the grid size, as shown in Figure 9.

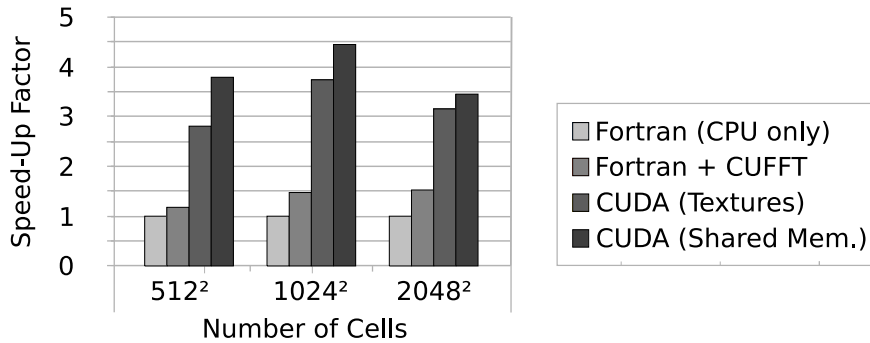


Fig. 9. Performance improvements using GPU acceleration. Speed-ups of the 2D meteorological application based on the number of discretization points in any direction.

3D particulate flow simulation: The efficient solution of the linear equation systems arising after discretization and linearization is a complex problem on its own. Usually either a multi-grid solver or a preconditioned conjugate gradients (CG) solver is employed, as it is done here. In each iteration of the CG algorithm one matrix vector product is needed, three scalar products and three vector updates in the form of an *axpy* operation. In a first step, all but the *csrmmv* operation can be taken from CUBLAS. As again only read operations are performed on the source matrix, we can accelerate the matrix-vector product

easily by making use of the GPU's texture caches. Comparing the resulting implementation to a similar one with calls to Intel's MKL (including its own *csrcmv*) we achieve already speed-ups of 2.5 to 3.2 at reasonable domain resolutions, as shown in Figure 10. For the implementation on the GPU it would be preferable to combine as much operations as possible into a single kernel, such that no register flush is done when executing the next function. In order to perform every scalar product operation in parallel and also all vector updates, the operations of the CG method have to be reordered. At first the problem arises, that due to non-solvable dependencies similar operations cannot be blocked. A solution to this was presented in [3] for a CRAY vector computer, that computes an additional scalar product but allows pipelining on the other hand. Using this method in a CUDA-only implementation leads to final speed-ups between 3 and 4.4 for relevant grid sizes.

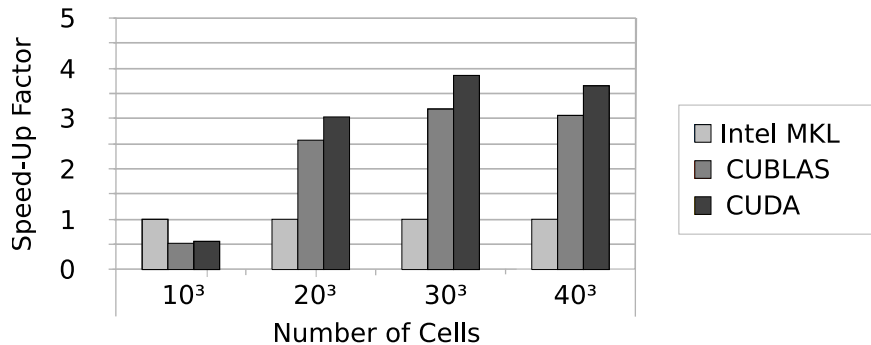


Fig. 10. Performance improvements using GPU acceleration. Runtime of the 2D meteorological application and 3D particulate flow simulation based on the number of discretization points in any direction.

5 Conclusion

The current GPU generation offers enormous potential that can be utilized not only in constructed examples but also CFD applications with academic and non-academic background. In our simulations we achieved speed-up factors of up to 4.5.

An essential condition is that the underlying mathematical model, combined with the numerical schemes for solving it, offers enough parallelism to create a sufficient number of threads for the GPU. Memory latency stalling one GPU-thread can be hidden by executing another thread. This is efficient, as exchanging threads is cheap compared to the costs of memory operations.

The programmability of NVIDIA-GPUs was heavily simplified by the introduction of CUDA compared to former approaches. Significant performance increases can be achieved in very short time e.g. by extending existing applications with accelerated kernels in Fortran and C/C++ as our partially ported meteorological implementation shows. To port a code completely to the GPU then offers the full performance of the architecture.

All this is available for a relatively low prize not only in form of consumer graphics cards (*GeForce* series) but also in professional solutions for high performance computing (*TESLA* series). By analyzing the TOP500 list [14] the number of clusters with accelerators, often NVIDIA GPUs, increases more and more and gives a strong impulse to make use of this technology.

References

1. NVIDIA Corp.: NVIDIA GeForce®GTX 200 GPU Architectural Overview. May, 2008. - URL http://www.nvidia.com/object/io_1213615494642.html.
2. NVIDIA Corp.: NVIDIA CUDA Programming Guide 2.0. June, 2008. - URL http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf.
3. Meurant, G.: Multitasking the conjugate gradient on the CRAY X-MP/48. Technical Report, Stanford University, CA, USA, 1985.
4. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics*, vol. 22, 2003, pp. 917-924.
5. Krüger, J., Westermann, R.: Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, vol. 22, 2003, pp. 908-916.
6. Goodnight, N., Lewin, G., Luebke, D., Skadron, K.: A multigrid solver for boundary-value problems using programmable graphics hardware. *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2003, pp. 102-111.
7. Hu, H.H., Joseph, D.D., Crochet, M.J.: Direct simulation of fluid particle motions. *Theoretical Computational Fluid Dynamics*, vol. 169, 1992, pp. 285-306.
8. Galdi, G.P., Heuveline, V.: Lift and sedimentation of particles in the flow of a viscoelastic liquid in a channel. *Lecture Notes in Pure and Applied Mathematics*, vol. 252. Chapman and Hall, 2007, pp. 75-110.
9. Glowinski, R., Pan, T.W., Hesla, T.I., Joseph, D.D.: A distributed Lagrange multiplier/fictitious domain method for particulate flows. *International Journal of Multiphase Flow*, vol. 25, 1999, pp. 755-794.
10. Veeramani, C. and Mineev, P.D. and Nandakumar, K.: A fictitious domain formulation for flows with rigid particles: A non-Lagrange multiplier version. *Journal of Computational Physics*, vol 224, 2007, pp. 867-879.
11. Hahn, T.: Numerical simulation of 3D particulate flows based on GPU technology. Diploma Thesis, Institute for Applied and Numerical Mathematics, University of Karlsruhe (TH). April, 2009.
12. Brown, R. A.: *Fluid Mechanics of the Atmosphere*. Academic Press, London (1992).
13. Ciarlet, P. G. and Lions, J. J.: *Numerical Methods for Fluids (Part 3)*. Handbook of Numerical Analysis vol. 9. North-Holland, 2003, pp. 23-26.
14. Homepage of the TOP500 project. URL <http://top500.org>
15. Majda, A. J. and Bertozzi, A. L.: *Vorticity and Incompressible Flow*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002, chapter 2.2.

A Universal Framework for Simulating Hierarchical Network Topologies in a Distributed Memory System

Oliver Mattes and Rainer Buchty

Universität Karlsruhe (TH),
Institute of Computer Science & Engineering (ITEC),
Chair for Computer Architecture,
Zirkel 2, 76131 Karlsruhe, Germany
`{mattes,buchty}@ira.uka.de`

Extended Abstract

Abstract. In distributed systems, the used interconnection network takes a vital position: the performance of the entire system and the adopted protocols depend highly on the underlying network structure. To identify the usability of several protocols on different network structures, a flexible simulation module is needed. In this paper we present a framework for simulating various kinds of network structures. It supports hierarchical structures and dynamically changing the network parameters at runtime. The benefit of this framework is demonstrated by an application example, in which the framework is embedded within a network module for the UNIted SIMulation environment (UNISIM) simulating protocol alternatives for the distributed memory architecture Self-aware Memory (SaM).

Key words: Simulation, Network, Memory System, Self-Aware Memory, UNISIM

1 Introduction

Future architectures are becoming increasingly complex and using these resources properly is turning into a major problem. Hence, novel approaches for future multicore architectures are currently being explored targeting more autonomous behavior in terms of resource management and use. One example of such future architectures is the Digital on-demand Computing Organism (DodOrg) [1] which comprises a set of reconfigurable processing cells interconnected by a flexible peer-to-peer network. One major issue in this architecture is providing a consistent, configuration-independent memory subsystem to the individual processing cells. Similar problems arise in future multicore architectures which not only use a heterogeneous mix of individual processor cores but also employ reconfigurable accelerator hardware: also for these architectures, memory management becomes an increasing problem especially with respect to aspects of heterogeneity and reconfigurability.

As a possible solution to this we introduced Self-aware Memory (SaM) [2,3], a distributed, scalable, and self-managing memory architecture for multi-master environments. To examine and test this architecture, a simulation infrastructure was implemented as an initial concept study using the UNISIM simulator framework [4]. In order to fully simulate the self-managing aspects of SaM, a network module is required which can be dynamically changed during runtime, so that for example the reactions of the whole system to changes of the network can be analyzed.

As it turned out, the UNISIM-provided network modules are not sufficient to easily evaluate such dynamic infrastructures and require efforts such as describing a super-set of all scenarios to be evaluated and dynamically switching between these. While this is possible for smaller setups, it is not feasible with bigger simulation scenarios, leading to high programming effort and runtime requirements.

For this purpose we developed a dedicated, flexible network module. In order to be universally usable and not restricted to the SaM and UNISIM context, it was implemented as an independent framework which can be integrated conveniently into own projects where a fast and simple but also dynamic and extendable network model is needed.

In the following section we will first provide a requirements analysis and comparison with existing approaches. We then present the concept of our independent network framework. After that, the implementation and results of the framework are provided, combined with a usage example where the framework is embedded into a UNISIM module.

2 Related Work

For our work, a highly configurable, dynamical, and flexible but lightweight network framework with easy-to-use functions is required. Offering a compatible timing model for the integration into a UNISIM module also has to be assured.

Several dedicated network simulators exist. An exhaustive overview over this field is given on the web page of the ICSI Networking Group [5]. Despite the fair amount of existing alternatives, none of these served our needs: typically, they are intended to simulate complete computer systems and network infrastructures and therefore act as a stand-alone application. While offering a huge collection of features, they are hardly extendable and integrable especially into the UNISIM framework because of different timing concepts. As examples for such simulator infrastructures, Ns-2 [6], the SSF Network Models (SSFNet) [7], and OMNeT++ [8] can be referred. Further commercial network simulations exist like OpNet, Netwiser, Shunra, and NetScale, which cannot be used due to license reasons.

We therefore decided to develop a simulation framework exactly fitting our requirements.

3 Concept of a universal network framework

The intention of this work was the development of a network module for UNISIM. During the design of this module however we decided to split the actual network functionality from the wrapping of a UNISIM module in order to not follow existing approaches, limiting the usage of the network simulation to our framework. In Figure 1 the basic structure of this module and the surrounding framework are shown.

The illustrated network module is connected to several other external modules, each linked to a network node by an appropriate interface. The modules are synchronized by a system-wide clock with constant clock cycles, so the network module has to use the same timing information. For exchanging data, messages representing packets of fixed length are passed between the modules on every cycle. The network structure is represented by a flexible internal model. Complex structures are separated into different topologies with a joint internal bridging node.

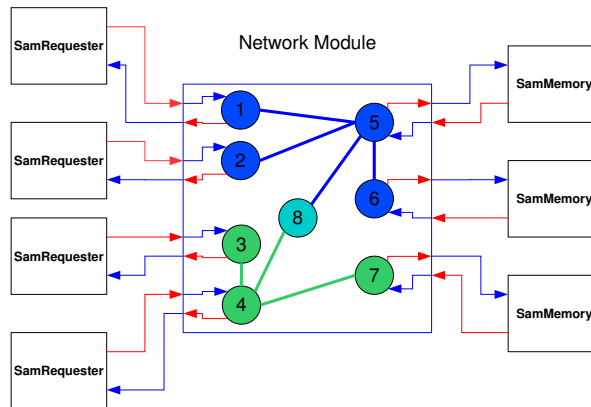


Fig. 1. Example for the structure and connections of a network module

The network framework has to fulfill certain requirements to suit our needs. First of all, the network simulation is a discrete simulation, therefore the *simulation time* has to be discrete. Within the framework, time is represented as a global event, occurring as a timing event in every instantiated module. Also the transferred data should be divided into discrete *information units*. To achieve this, an information unit is represented by a message. The amount of information can then be measured by the number of messages sent. The *network model* should be highly flexible. To achieve this, more complicated network structures can be split into several topologies. In these topologies, nodes and edges may exist. An edge always has to be connected to exactly two nodes. For connecting several topologies, inner nodes called EdgeNode exist which act as a bridge between two topologies, e.g. in Figure 1 node #8. This way, highly flexible and dynamic structures of the network and networks consisting of completely different structures can be built. To each edge, latency and communication bandwidth are assigned as parameters. Latency is represented by the amount of time events that a message takes to be transferred over this edge. This is similar to the distance between two nodes. In this model, the bandwidth is the amount of information units which can be transferred over the edge at the same time. For the simulation of scalable networks, efficient routing algorithms are required. Depending on the size of the network, different methods have to be used because the search for the shortest path is limited by memory size and computing time. To handle this problem, routing management is split up between the individual topologies, potentially enabling parallel execution.

4 Implementation and Results

In Section 3, the concept of the network framework was presented. The announced partitioning into several parts is realized in the implementation by a compound of five C++ classes, the dependencies between these classes are shown in Figure 2. The full paper will give more detailed information about the implementation.

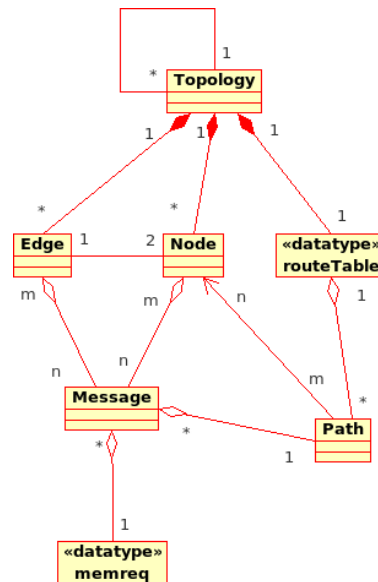


Fig. 2. UML structure of the network framework

The framework was tested for correctness and scalability, the latter with tests rising up to network structures with more than 10000 nodes, where sending and receiving of individual messages was still possible. High workload with concurrent transfers between all nodes was also tested.

The framework can be easily embedded into UNISIM. The communication with other modules is obtained through proper interfaces: Listing 1.1 shows the creation of a small topology with a number of nodes and edges. The network description can be saved into a configuration file and reloaded at the beginning of the simulation as shown in Listing 1.2, also showing a dynamic change of the weight of an edge during runtime.

Using this framework within UNISIM, evaluations of new protocols and data transfers for SaM are currently performed.

```

1 // fragments from main.cpp
2
3 // creating three topology objects
4 Topology tp1, tp2, tp3;
5
6 // creating nodes in the topology
7 // object tp1
8 for (int i= 0; i<5; i++)
9     tp1.addNewNode();
10
11 // creating edges between the nodes
12 tp1.addNewEdge(1,3,4);
13 tp1.addNewEdge(2,3,2);
14 tp1.addNewEdge(4,3,1);
15 tp1.addNewEdge(5,3,2);

```

Listing 1.1. Creating a Network

```

1 // taken from the file Network.sim
2
3 // Topology Object declared as private
4 Topology vTopology;
5
6 // in the constructor of the module
7 // load the configuration of the network
8 vTopology.loadFromFile("Network.conf");
9
10 //...
11
12 // changing the weight of edge
13 // between node 3 and 6
14 if (k == 61)
15 {
16     Edge * tempEdge;
17     tempEdge = vTopology.getEdge(3,6);
18     tempEdge->setWeight(100);
19 }

```

Listing 1.2. Changing the Bandwidth

5 Conclusion and Outlook

Starting with a short introduction in network simulations we motivated a scalable, dynamical, and highly flexible network framework for simulating arbitrary network structures consisting of a high number of connected nodes.

By separating the network functionality from the surrounding UNISIM module, a generic network framework is offered which can be integrated into own designs using the implemented interfaces and according service primitives. Extensions for graphical representations of the network and protocols for safe transfers are intended and can be easily added.

As of now, the framework supports changing of network topologies and parameters at runtime and it is easily integrable into existing simulation frameworks such as UNISIM.

References

1. Becker, J., Brändle, K., Brinkschulte, U., Henkel, J., Karl, W., Köster, T., Wenz, M., Wörn, H.: Digital On-Demand Computing Organism for Real-Time Systems. In Karl, W., Becker, J., Großpietsch, K.E., Hochberger, C., Maehle, E., eds.: Workshop Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS'06). Volume P81 of GI-Edition Lecture Notes in Informatics (LNI). (March 2006) 230–245
2. Buchty, R., Mattes, O., Karl, W.: Self-aware Memory: Managing Distributed Memory in an Autonomous Multi-Master Environment. In: Architecture of Computing Systems – ARCS 2008, 21st International Conference. Volume Series Lecture Notes in Computer Science (LNCS), Volume 4934., Dresden, Germany (February 25-28 2008) 98–113
3. Mattes, O.: Developement of a decentral memory management for distrubuted systems (Self-aware Memory). Master's thesis, University of Karlsruhe (2007)
4. European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC): UNISIM: UNIted SIMulation Environment
5. The ICSI Networking Group, Sally Floyd: Network Simulators (September 2007) <http://www.icir.org/models/simulators.html>.
6. The Network Simulator: Ns-2 (March 2009) <http://nsnam.isi.edu/nsnam/>.
7. ssfnet.org: Scalable SIMulation Framework SSFNet (March 2009) <http://www.ssfnet.org/>.
8. OMNet++: Discrete Event Simulation System (March 2009) <http://www.omnetpp.org/>.

1. Aktuelle und zukünftige Aktivitäten (Bericht des Sprechers)

Die 26. Ausgabe der PARS-Mitteilungen enthält die Beiträge des 22. PARS-Workshops, welcher die wesentlichen Aktivitäten der Fachgruppe im Jahr 2009 darstellt.

Der 22. PARS-Workshop fand am 4./5. Juni 2009 in Parsberg in der Oberpfalz statt. Fast 35 Teilnehmer fanden sich an ein. Die zehn Vorträge deckten ein umfangreiches Themenspektrum ab.

Den zum vierten Mal ausgeschriebenen und mit 500 € dotierten Nachwuchspreis teilen sich in diesem Jahr Frau Annika Schiller (FZ Jülich) und Herr Christian Schäck (TU Darmstadt). Sachpreise konnten Andre Luckow (Universität Potsdam) und Stanimir Dragiev (TU Berlin) entgegennehmen. Das Bild zeigt die Preisträger zusammen mit den Organisatoren des Workshops beim abendlichen Social-Event.



v.l.n.r.: Prof. Dr. Wolfgang Karl (Organisation), Christian Schäck, Andre Luckow, Stanimir Dragiev, Annika Schiller, Prof. Dr. Rolf Hoffmann (Sprecher der Fachgruppe)

Bild: K. D. Reinartz

Herrn Prof. Dr. Wolfgang Karl (Universität Karlsruhe) sei für die federführende Organisation und die lokale Gestaltung des Workshops gedankt. Herrn Dr.-Ing. Wolfgang Heenes (TU Darmstadt) sei für die technische Organisation gedankt.

Im Anschluss an den PARS-Workshop fand eine Sitzung des PARS-Leitungsgremiums statt.



v.l.n.r.: Prof. Dr. H. Weberpals, Dr. K.D. Reinartz, Prof. Dr. J. Keller, Dr. A. Döring, Prof. Dr. R. Hoffmann, Prof. Dr. W. Karl

Bild: Brehm

Zum Nachfolger des Sprechers ab 2010 wurde Herr Prof. Dr. Jörg Keller gewählt. Als stellvertretender Sprecher wurde Herr Dr. Karl Dieter Reinartz in seinem Amt bestätigt.

Unser nächster Workshop ist der

23. PARS-Workshop am 23. Februar 2010 in Hannover im Rahmen der ARCS 2010.

Die nächste Sitzung des PARS-Leitungsgremiums, zu dem auch interessierte PARS-Mitglieder eingeladen sind, wird am Rande des PARS-Workshops 2010 in Hannover stattfinden.

Aktuelle Information finden Sie auch auf der PARS-Webpage

<http://www.fg-pars.gi-ev.de/>

Anregungen und Beiträge für die Mitteilungen können an den Sprecher (hoffmann@ra.informatik.tu-darmstadt.de bzw. joerg.keller@FernUni-Hagen.de) gesendet werden. Besonders gedankt sei meinem Mitarbeiter Herrn Dr. Heenes, der mich in technischen und allgemeinen Verwaltungsaufgaben unterstützt. Er hat diese PARS-Mitteilungen für den Druck zusammengestellt und gestaltet u. a. den Web-Auftritt von PARS.

Ich wünsche Ihnen ein gesundes und erfolgreiches Jahr 2010.

Darmstadt, im Dezember 2009

R. Hoffmann

2. Zur Historie von PARS

Bereits am Rande der Tagung CONPAR81 vom 10. bis 12. Juni 1981 in Nürnberg wurde von Teilnehmern dieser ersten CONPAR-Veranstaltung die Gründung eines Arbeitskreises im Rahmen der GI: Parallel-Algorithmen und -Rechnerstrukturen angeregt. Daraufhin erfolgte im Heft 2, 1982 der GI-Mitteilungen ein Aufruf zur Mitarbeit. Dort wurden auch die Themen und Schwerpunkte genannt:

1) Entwurf von Algorithmen für

- verschiedene Strukturen (z. B. für Vektorprozessoren, systolische Arrays oder Zellprozessoren)
- Verifikation
- Komplexitätsfragen

2) Strukturen und Funktionen

- Klassifikationen
- dynamische/rekonfigurierbare Systeme
- Vektor/Pipeline-Prozessoren und Multiprozessoren
- Assoziative Prozessoren
- Datenflussrechner
- Reduktionsrechner (demand driven)
- Zellulare und Systolische Systeme
- Spezialrechner, z. B. Baumrechner und Datenbank-Prozessoren

3) Intra-Kommunikation

- Speicherorganisation
- Verbindungsnetzwerke

4) Wechselwirkung zwischen paralleler Struktur und Systemsoftware

- Betriebssysteme
- Compiler

5) Sprachen

- Erweiterungen (z. B. für Vektor/Pipeline-Prozessoren)
- (automatische) Parallelisierung sequentieller Algorithmen
- originär parallele Sprachen
- Compiler

6) Modellierung, Leistungsanalyse und Bewertung

- theoretische Basis (z. B. Q-Theorie)
- Methodik
- Kriterien (bezüglich Strukturen)
- Analytik

In der Sitzung des Fachbereichs 3 ‚Architektur und Betrieb von Rechensystemen‘ der Gesellschaft für Informatik am 22. Februar 1983 wurde der Arbeitskreis offiziell gegründet. Nachdem die Mitgliederzahl schnell anwuchs, wurde in der Sitzung des Fachausschusses 3.1 ‚Systemarchitektur‘ am 20. September 1985 in Wien der ursprüngliche Arbeitskreis in die Fachgruppe FG 3.1.2 ‚Parallel- Algorithmen und - Rechnerstrukturen‘ umgewandelt.

Während eines Workshops vom 12. bis 16. Juni 1989 in Rurberg (Aachen) - veranstaltet von den Herren Ecker (TU Clausthal) und Lange (TU Hamburg-Harburg) - wurde vereinbart, Folgeveranstaltungen hierzu künftig im Rahmen von PARS durchzuführen.

Beim Workshop in Arnoldshain sprachen sich die PARS-Mitglieder und die ITG-Vertreter dafür aus, die Zusammenarbeit fortzusetzen und zu verstärken. Am Dienstag, dem 20. März 1990 fand deshalb in

München eine Vorbesprechung zur Gründung einer gemeinsamen Fachgruppe PARS statt. Am 6. Mai 1991 wurde in einer weiteren Besprechung eine Vereinbarung zwischen GI und ITG sowie eine Vereinbarung und eine Ordnung für die gemeinsame Fachgruppe PARS formuliert und den beiden Gesellschaften zugeleitet. Die GI hat dem bereits 1991 und die ITG am 26. Februar 1992 zugestimmt.

3. Bisherige Aktivitäten

Die PARS-Gruppe hat in den vergangenen Jahren mehr als 20 Workshops durchgeführt mit Berichten und Diskussionen zum genannten Themenkreis aus den Hochschulinstituten, Großforschungseinrichtungen und der einschlägigen Industrie. Die Industrie - sowohl die Anbieter von Systemen wie auch die Anwender mit speziellen Problemen - in die wissenschaftliche Erörterung einzubeziehen war von Anfang an ein besonderes Anliegen. Durch die immer schneller wachsende Zahl von Anbietern paralleler Systeme wird sich die Mitgliederzahl auch aus diesem Kreis weiter vergrößern.

Neben diesen Workshops hat die PARS-Gruppe die örtlichen Tagungsleitungen der CONPAR-Veranstaltungen:

CONPAR 86 in Aachen,
CONPAR 88 in Manchester,
CONPAR 90 / VAPP IV in Zürich und
CONPAR 92 / VAPP V in Lyon
CONPAR 94/VAPP VI in Linz

wesentlich unterstützt. In einer Sitzung am 15. Juni 1993 in München wurde eine Zusammenlegung der Parallelrechner-Tagungen von CONPAR/VAPP und PARLE zur neuen Tagungsserie EURO-PAR vereinbart, die vom 29. bis 31. August 1995 erstmals stattfand:

Euro-Par'95 in Stockholm

Zu diesem Zweck wurde ein „Steering Committee“ ernannt, das europaweit in Koordination mit ähnlichen Aktivitäten anderer Gruppierungen Parallelrechner-Tagungen planen und durchführen wird. Dem Steering Committee steht ein „Advisory Board“ mit Personen zur Seite, die sich in diesem Bereich besonders engagieren. Die offizielle Homepage von Euro-Par ist <http://www.euro-par.org/>.

Weitere bisher durchgeführte Veranstaltungen:

Euro-Par'96 in Lyon
Euro-Par'97 in Passau
Euro-Par'98 in Southampton
Euro-Par'99 in Toulouse
Euro-Par 2000 in München
Euro-Par 2001 in Manchester
Euro-Par 2002 in Paderborn
Euro-Par 2003 in Klagenfurt
Euro-Par 2004 in Pisa
Euro-Par 2005 in Lissabon
Euro-Par 2006 in Dresden
Euro-Par 2007 in Rennes
Euro-Par 2008 in Gran Canaria
Euro-Par 2009 in Delft

Außerdem war die Fachgruppe bemüht, mit anderen Fachgruppen der Gesellschaft für Informatik übergreifende Themen gemeinsam zu behandeln: Workshops in Bad Honnef 1988, Dagstuhl 1992 und Bad Honnef 1996 (je zusammen mit der FG 2.1.4 der GI), in Stuttgart (zusammen mit dem Institut für Mikroelektronik) und die PASA-Workshop-Reihe 1991 in Paderborn, 1993 in Bonn, 1996 in Jülich, 1999 in Jena, 2002 in Karlsruhe, 2004 in Augsburg (jeweils gemeinsam mit der GI-Fachgruppe 0.1.3 „Parallele und verteilte Algorithmen (PARVA)“).

PARS-Mitteilungen/Workshops:

- Aufruf zur Mitarbeit, April 1983 (Mitteilungen Nr. 1)
Erlangen, 12./13. April 1984 (Mitteilungen Nr. 2)
Braunschweig, 21./22. März 1985 (Mitteilungen Nr. 3)
Jülich, 2./3. April 1987 (Mitteilungen Nr. 4)
Bad Honnef, 16.-18. Mai 1988 (Mitteilungen Nr. 5, gemeinsam mit der GI-Fachgruppe 2.1.4 'Alternative Konzepte für Sprachen und Rechner')
München Neu-Perlach, 10.-12. April 1989 (Mitteilungen Nr. 6)
Arnoldshain (Taunus), 25./26. Januar 1990 (Mitteilungen Nr. 7)
Stuttgart, 23./24. September 1991, "Verbindungsnetzwerke für Parallelrechner und Breitband-Übermittlungssysteme" (Als Mitteilungen Nr. 8 geplant, gemeinsam mit ITG-FA 4.1 und 4.4 und mit GI/ITG FG Rechnernetze, aber aus Kostengründen nicht erschienen. Es wird deshalb stattdessen auf den Tagungsband des Instituts für Mikroelektronik Stuttgart hingewiesen.)
Paderborn, 7./8. Oktober 1991, "Parallele Systeme und Algorithmen" (Mitteilungen Nr. 9, 2. PASA-Workshop)
Dagstuhl, 26.-28. Februar 1992, "Parallelrechner und Programmiersprachen" (Mitteilungen Nr. 10, gemeinsam mit der GI-Fachgruppe 2.1.4 'Alternative Konzepte für Sprachen und Rechner')
Bonn, 1./2. April 1993, "Parallele Systeme und Algorithmen" (Mitteilungen Nr. 11, 3. PASA-Workshop)
Dresden, 6.-8. April 1993, "Feinkörnige und Massive Parallelität" (Mitteilungen Nr. 12, zusammen mit PARCELLA)
Potsdam, 19./20. September 1994 (Mitteilungen Nr. 13, Parcella fand dort anschließend statt)
Stuttgart, 9.-11. Oktober 1995 (Mitteilungen Nr. 14)
Jülich, 10.-12. April 1996, "Parallel Systems and Algorithms" (4. PASA-Workshop), Tagungsband erschienen bei World Scientific 1997)
Bad Honnef, 13.-15. Mai 1996, zusammen mit der GI-Fachgruppe 2.1.4 'Alternative Konzepte für Sprachen und Rechner' (Mitteilungen Nr. 15)
Rostock, (Warnemünde) 11. September 1997 (Mitteilungen Nr. 16, im Rahmen der ARCS'97 vom 8.-11. September 1997)
Karlsruhe, 16.-17. September 1998 (Mitteilungen Nr. 17)
Jena, 7. September 1999, "Parallele Systeme und Algorithmen" (5. PASA-Workshop im Rahmen der ARCS'99)
An Stelle eines Workshop-Bandes wurde den PARS-Mitgliedern im Januar 2000 das Buch 'SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Compute Clusters', Hermann Hellwagner und Alexander Reinefeld (Eds.) zur Verfügung gestellt.
München, 8.-9. Oktober 2001 (Mitteilungen Nr. 18)
Karlsruhe, 11. April 2002, "Parallele Systeme und Algorithmen" (Mitteilungen Nr. 19, 6. PASA-Workshop im Rahmen der ARCS 2002)
Travemünde, 5./6. Juli 2002, Brainstorming Workshop "Future Trends" (Thesenpapier in Mitteilungen Nr. 19)
Basel, 20./21. März 2003 (Mitteilungen Nr. 20)
Augsburg, 26. März 2004 (Mitteilungen Nr. 21)
Lübeck, 23./24. Juni 2005 (Mitteilungen Nr. 22)
Frankfurt/Main, 16. März 2006 (Mitteilungen Nr. 23)
Hamburg, 31. Mai / 1. Juni 2007 (Mitteilungen Nr. 24)
Dresden, 26. Februar 2008 (Mitteilungen Nr. 25)
Parsberg, 4./5. Juni 2009 (Mitteilungen Nr. 26)

4. Mitteilungen (ISSN 0177-0454)

Bisher sind 25 Mitteilungen zur Veröffentlichung der PARS-Aktivitäten und verschiedener Workshops erschienen. Darüberhinaus enthalten die Mitteilungen Kurzberichte der Mitglieder und Hinweise von allgemeinem Interesse, die dem Sprecher zugetragen werden.

Teilen Sie - soweit das nicht schon geschehen ist - Tel., Fax und E-Mail-Adresse der GI-Geschäftsstelle gimv@gi-ev.de mit für die zentrale Datenerfassung und die regelmäßige Übernahme in die PARS-Mitgliederliste. Das verbessert unsere Kommunikationsmöglichkeiten untereinander wesentlich.

5. Vereinbarung

Die Gesellschaft für Informatik (GI) und die Informationstechnische Gesellschaft im VDE (ITG) vereinbaren die Gründung einer gemeinsamen Fachgruppe

Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware,

die den GI-Fachausschüssen bzw. Fachbereichen:

FA 0.1	Theorie der Parallelverarbeitung
FA 3.1	Systemarchitektur
FB 4	Informationstechnik und technische Nutzung der Informatik

und den ITG-Fachausschüssen:

FA 4.1	Rechner- und Systemarchitektur
FA 4.2/3	System- und Anwendungssoftware

zugeordnet ist.

Die Gründung der gemeinsamen Fachgruppe hat das Ziel,

- die Kräfte beider Gesellschaften auf dem genannten Fachgebiet zusammenzulegen,
- interessierte Fachleute möglichst unmittelbar die Arbeit der Gesellschaften auf diesem Gebiet gestalten zu lassen,
- für die internationale Zusammenarbeit eine deutsche Partnergruppe zu haben.

Die fachliche Zielsetzung der Fachgruppe umfasst alle Formen der Parallelität wie

- Nebenläufigkeit
- Pipelining
- Assoziativität
- Systolik
- Datenfluss
- Reduktion
- etc.

und wird durch die untenstehenden Aspekte und deren vielschichtige Wechselwirkungen umrissen. Dabei wird davon ausgegangen, dass in jedem der angegebenen Bereiche die theoretische Fundierung und Betrachtung der Wechselwirkungen in der Systemarchitektur eingeschlossen ist, so dass ein gesonderter Punkt „Theorie der Parallelverarbeitung“ entfällt.

1. Parallelrechner-Algorithmen und -Anwendungen

- architekturabhängig, architekturunabhängig
- numerische und nichtnumerische Algorithmen
- Spezifikation
- Verifikation
- Komplexität
- Implementierung

2. Parallelrechner-Software

- Programmiersprachen und ihre Compiler
- Programmierwerkzeuge
- Betriebssysteme

3. Parallelrechner-Architekturen

- Ausführungsmodelle
- Verbindungsstrukturen
- Verarbeitungselemente
- Speicherstrukturen
- Peripheriestrukturen

4. Parallelrechner-Modellierung, -Leistungsanalyse und -Bewertung

5. Parallelrechner-Klassifikation, Taxonomien

Als Gründungsmitglieder werden bestellt:

von der GI: Prof. Dr. A. Bode, Prof. Dr. W. Gentzsch, R. Kober, Prof. Dr. E. Mayr, Dr. K. D. Reinartz, Prof. Dr. P. P. Spies, Prof. Dr. W. Händler

von der ITG: Prof. Dr. R. Hoffmann, Prof. Dr. P. Müller-Stoy, Dr. T. Schwederski, Prof. Dr. Swoboda, G. Valdorf

Ordnung der Fachgruppe

Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware

1. Die Fachgruppe wird gemeinsam von den Fachausschüssen 0.1, 3.1 sowie dem Fachbereich 4 der Gesellschaft für Informatik (GI) und von den Fachausschüssen 4.1 und 4.2/3 der Informationstechnischen Gesellschaft (ITG) geführt.
2. Der Fachgruppe kann jedes interessierte Mitglied der beteiligten Gesellschaften beitreten. Die Fachgruppe kann in Ausnahmefällen auch fachlich Interessierte aufnehmen, die nicht Mitglied einer der beteiligten Gesellschaften sind. Mitglieder der FG 3.1.2 der GI und der ITG-Fachgruppe 6.1.2 werden automatisch Mitglieder der gemeinsamen Fachgruppe PARS.
3. Die Fachgruppe wird von einem ca. zehnköpfigen Leitungsgremium geleitet, das sich paritätisch aus Mitgliedern der beteiligten Gesellschaften zusammensetzen soll. Für jede Gesellschaft bestimmt deren Fachbereich (FB 3 der GI und FB 4 der ITG) drei Mitglieder des Leitungsgremiums: die übrigen werden durch die Mitglieder der Fachgruppe gewählt. Die Wahl- und die Berufungsvorschläge macht das Leitungsgremium der Fachgruppe. Die Amtszeit der Mitglieder des Leitungsgremiums beträgt vier Jahre. Wiederwahl ist zulässig.
4. Das Leitungsgremium wählt aus seiner Mitte einen Sprecher und dessen Stellvertreter für die Dauer von zwei Jahren; dabei sollen beide Gesellschaften vertreten sein. Wiederwahl ist zulässig. Der Sprecher führt die Geschäfte der Fachgruppe, wobei er an Beschlüsse des Leitungsgremiums gebunden ist. Der Sprecher besorgt die erforderlichen Wahlen und amtiert bis zur Wahl eines neuen Sprechers.
5. Die Fachgruppe handelt im gegenseitigen Einvernehmen mit den genannten Fachausschüssen. Die Fachgruppe informiert die genannten Fachausschüsse rechtzeitig über ihre geplanten Aktivitäten. Ebenso informieren die Fachausschüsse die Fachgruppe und die anderen beteiligten Fachausschüsse über Planungen, die das genannte Fachgebiet betreffen. Die Fachausschüsse unterstützen die Fachgruppe beim Aufbau einer internationalen Zusammenarbeit und stellen ihr in angemessenem Umfang ihre Publikationsmöglichkeiten zur Verfügung. Die Fachgruppe kann keine die Trägergesellschaften verpflichtenden Erklärungen abgeben.
6. Veranstaltungen (Tagungen/Workshops usw.) sollten abwechselnd von den Gesellschaften organisiert werden. Kostengesichtspunkte sind dabei zu berücksichtigen.
7. Veröffentlichungen, die über die Fachgruppenmitteilungen hinausgehen, z. B. Tagungsberichte, sollten in Abstimmung mit den den Gesellschaften verbundenen Verlagen herausgegeben werden. Bei den Veröffentlichungen soll ein durchgehend einheitliches Erscheinungsbild angestrebt werden.
8. Die gemeinsame Fachgruppe kann durch einseitige Erklärung einer der beteiligten Gesellschaften aufgelöst werden. Die Ordnung tritt mit dem Datum der Unterschrift unter die Vereinbarung über die gemeinsame Fachgruppe in Kraft.

CALL FOR PAPERS

23rd PARS-Workshop on Parallel Systems and Algorithms PARS 2010

in conjunction with

23rd International Conference on Architecture of Computing Systems (ARCS)
Hannover, Germany, February 22nd to 25th, 2010

organized by

GI/ITG-Fachgruppe 'Parallel-Algorithmen, - Rechnerstrukturen und - Systemsoftware' (PARS)

The PARS-Workshop series has the goal to build a bridge between theory and practice in the area of parallel systems and algorithms. In this context practical problems which require theoretical investigations as well as the applicability of theoretical approaches and results to practice shall be discussed. An important aspect is communication and exchange of experience between various groups working in the area of parallel computing, e.g. in computer science, electrical engineering, physics or mathematics.

Topics of Interest include, but are not restricted to:

- parallel architectures & storage systems
- parallel embedded systems
- ubiquitous and pervasive systems
- models of parallel computation
- data stream-oriented computing
- parallel and distributed algorithms
- network and grid computing
- parallel organic computing
- interconnection networks
- reconfigurable parallel computing
- distributed and parallel multimedia systems
- performance evaluation of parallel systems
- software engineering for parallel systems
- parallel programming languages
- new technologies & architectures (SoC, Multicores, PIM, etc.)
- alternative technologies (e.g. quantum or DNA computing)

The workshop will comprise invited papers on current topics of leading experts in the field as well as submitted papers. The accepted papers shall have a length of about 10 pages A4 and will be published in the ARCS Workshop Proceedings as well as in the PARS Newsletter (ISSN 0177-0454). For ARCS 2010 participants the workshop is included in the conference fee, for participation in the workshop alone, we try to arrange a reduced fee. The conference language is English. Papers are required to be in English.

A prize of 500 € will be awarded to the best contribution presented personally by a student based on his or her MSc or PhD work. Co-authors are allowed. PhD shall not be granted upon submission. Application for the prize must be indicated upon submission. Applicants should become members of the Special Interest Group PARS.

Important Dates

October 15, 2009: Deadline for electronic submission of full papers of about 10 pages or extended abstracts of about 3 to 4 pages length under: <http://pv.fernuni-hagen.de/pars/2010/>. Papers must be submitted in pdf or postscript. They should be formatted according to Springer LNCS style (see: <http://www.springer.de/comp/lncs/authors.html>).

November 15, 2009: Notification of the authors

December 15, 2009: Final version for workshop proceedings

February 22/23, 2010: Workshop

Program Committee

H. Burkhart (Basel), A. Döring (Zürich), D. Fey (Erlangen), R. Hoffmann (Darmstadt), F. Hoßfeld (Jülich), W. Karl (Karlsruhe), J. Keller (Hagen), Ch. Lengauer (Passau), E. Maehle (Lübeck), E. W. Mayr (München), F. Meyer auf der Heide (Paderborn), W. E. Nagel (Dresden), K. D. Reinartz (Höchstadt), U. Rüde (Erlangen), H. Schmeck (Karlsruhe), U. Schwiegelshohn (Dortmund), P. Sobe (Lübeck), T. Ungerer (Augsburg), R. Wanka (Erlangen), H. Weberpals (Hamburg-Harburg)

Organisation

Prof. Dr. Jörg Keller, FernUniversität Hagen, Dept. Of Mathematics and Computer Science, Parallelism and VLSI Group, 58084 Hagen, Germany, Phone: +49-2331-987-376, Fax: +49-2331-987-308, E-Mail: joerg.keller@FernUni-Hagen.de

23RD INTERNATIONAL CONFERENCE ON ARCHITECTURE OF COMPUTING SYSTEMS**- HETEROGENEOUS SYSTEMS -**

Hannover, Germany

February 22nd - 25th, 2010<http://www.arcs2010.de>**CALL FOR PAPERS****EXTENDED SUBMISSION DEADLINE: September 27, 2009**

The ARCS series of conferences has over 30 years of tradition reporting top notch results in computer architecture and operating systems research. This year's focus will be on Heterogeneous Systems. Like the previous conferences in this series, it continues to be an important forum for computer architecture research. In 2010 ARCS will be hosted by the Leibniz University of Hannover.

The proceedings of ARCS 2010 will be published in the Springer Lecture Notes on Computer Science (LNCS) series. After the conference, authors of selected papers will be invited to submit an extended version of their contribution for publication in a special issue of the Journal of Systems Architecture. Also, a best paper and best presentation award will be presented at the conference.

Paper submission Authors are invited to submit original, unpublished research papers on one of the following topics:

- Computer architecture topics such as multi-cores, memory systems, and parallel computing.
- Adaptive system architectures such as reconfigurable systems in hardware and software.
- Customization and application specific accelerators in heterogeneous architectures
- Organic and Autonomic Computing including both theoretical and practical results on self-organization, self-configuration, self-optimization, self-healing, and self-protection techniques.
- Operating Systems including but not limited to scheduling, memory management, power management, and RTOS.
- Energy-awareness, green computing.
- System aspects of ubiquitous and pervasive computing such as sensor nodes, novel input/output devices, novel computing platforms, architecture modeling, and middleware.
- Embedded systems including but not limited to architecture, communication, design methodologies, and applications.
- Network Centric and Grid Computing

Submissions should be done through the link provided at the conference website <http://www.arcs2010.de/>. Papers should be submitted in pdf or postscript format. They should be formatted according to Springer LNCS style (see: <http://www.springer.de/comp/lncs/authors.html>) and not exceed 12 pages.

Workshop and Tutorial Proposals: Proposals for workshops and tutorials within the technical scope of the conference are solicited. Submissions should be done through email directly to the workshops and tutorials chair Michael Beigl (beigl@ibr.cs.tu-bs.de).

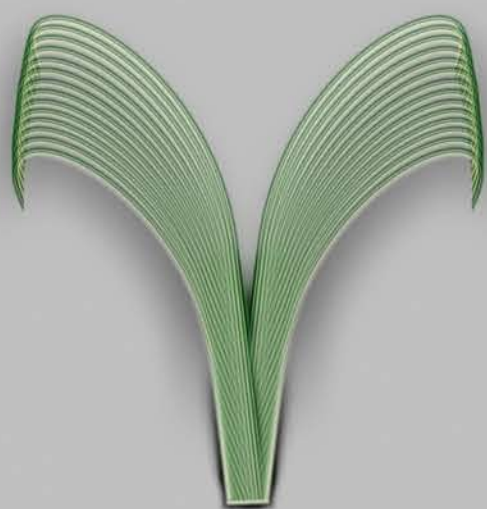
Important Dates

Extended Paper submission deadline: September 27, 2009

Workshop and tutorial proposals: October 15, 2009

Notification of acceptance: November 15, 2009

Camera ready papers: December 15, 2009



Euro-Par 2010

August 31st – September 3rd, 2010
Ischia - Naples, Italy

Key Dates

January 31, 2010	Abstracts due
February 7, 2010	Papers due
March 1, 2010	Workshop proposals due
May 7, 2010	Decision notification
June 1, 2010	Camera-ready full papers

Topics

1. Support Tools and Environments
2. Performance Prediction and Evaluation
3. Scheduling and Load-Balancing
4. High Performance Architectures and Compilers
5. Parallel and Distributed Data Management
6. Grid, Cluster and Cloud Computing
7. Peer to Peer Computing
8. Distributed Systems and Algorithms
9. Parallel and Distributed Programming
10. Parallel Numerical Algorithms
11. Multicore and Manycore Programming
12. Theory and Algorithms for Parallel Computation
13. High Performance Networks
14. Mobile and Ubiquitous Computing

Conference Workshops

Euro-Par 2010 will feature a series of satellite workshops on August 30th and 31st. Proposals for workshops covering a specific theme and lasting between half a day and a full day are encouraged and solicited until March 1st, 2010.

Venue

The Conference will take place at the Congress Center of the Hotel Continental Terme, in Ischia. Ischia, also known as the “Green Island”, is the largest island in the Gulf of Naples, to which it is connected with regular hydrofoils and ferries.

More information

<http://www.europar2010.it>
info@europar2010.it



Organization

Conference Co-Chairs

Domenico Talia

ICAR-CNR and University of Calabria

Pasqua D'Ambra

ICAR- CNR

Mario Rosario Guarracino

ICAR-CNR



*Institute for High-Performance
Computing and Networking*

PARS-Beiträge

Studenten	5,00 €
GI-Mitglieder	7,50 €
studentische Nichtmitglieder	5,00 €
Nichtmitglieder	15,00 €
Nichtmitglieder mit Doppel-Mitgliedschaften	
(Beitrag wie GI-Mitglieder)	--,-- €

Leitungsgremium von GI/ITG-PARS

Prof. Dr. Helmar Burkhart, Univ. Basel
Dr. Andreas Döring, IBM Zürich
Prof. Dr. Dietmar Fey, Univ. Erlangen
Prof. Dr. Rolf Hoffmann, Sprecher, TU Darmstadt
Prof. Dr. Wolfgang Karl, Univ. Karlsruhe
Prof. Dr. Jörg Keller, FernUniversität Hagen
Prof. Dr. Christian Lengauer, Univ. Passau
Prof. Dr.-Ing. Erik Maehle, Universität zu Lübeck
Prof. Dr. Ernst W. Mayr, TU München
Prof. Dr. Wolfgang E. Nagel, TU Dresden
Dr. Karl Dieter Reinartz, stellv. Sprecher, Univ. Erlangen-Nürnberg
Prof. Dr. Hartmut Schmeck, Univ. Karlsruhe
Prof. Dr. Theo Ungerer, Univ. Augsburg
Prof. Dr. Helmut Weberpals, TU Hamburg Harburg

Sprecher

Prof. Dr.-Ing. Rolf Hoffmann
Technische Universität Darmstadt
Fachgebiet Rechnerarchitektur
Hochschulstraße 10
D-64289 Darmstadt
Tel.: (06151) 16-3611/3606
Fax: (06151) 165410
E-Mail: hoffmann@informatik.tu-darmstadt.de
URL: <http://www.fg-pars.gi-ev.de/>

PARS
Einladung zur Mitarbeit
in der GI/ITG-Fachgruppe



Gesellschaft für Informatik e. V., Tel. (0228) 302145, FAX (0228) 302167, E-Mail: gs@gi-ev.de

- ☐ GI-Mitglied Nr. _____
- ☐ ITG-Mitglied Nr. _____
- ☐ Sonstige _____

- ☐ Ich beantrage die Mitgliedschaft in der Fachgruppe PARS
,Parallel-Algorithmen, -Rechnerstrukturen
und Systemsoftware'.
Ich bin bereit, meinen entsprechenden PARS-Beitrag
an die Geschäftsstelle der GI zu entrichten.

Gesellschaft für Informatik e.V.
Wissenschaftszentrum
Ahrstr. 45

D-53175 Bonn
FRG

Titel,	Name,	Vorname
Firma/Dienststelle		
Straße		
PLZ,	Ort,	Land
Telefon,	Datum,	Unterschrift
Telefax.	Email-Adresse	



**I
T
G**

**P
A
R
S**

'09