Hardware-aware solvers for large, sparse linear systems

Multi-precision and parallel approaches

Zur Erlangung des akademischen Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

der Fakultät für Mathematik der
Universität Karlsruhe (TH) genehmigte

DISSERTATION

von

Björn Henning Karl Rocker

aus Wilhelmshaven

Tag der mündlichen Prüfung: 17. Februar 2011

Referent: Prof. Dr. Vincent Heuveline
Korreferent: Prof. Dr. Götz Alefeld

Vorwort

Diese Dissertationsschrift entstand im Rahmen meiner Tätigkeit am Institut für Angewandte und Numerische Mathematik 4 (NumHPC) und dem Engineering Mathematics and Computing Lab (EMCL). Beide Einrichtungen sind Teil des Karlsruher Institut für Technologie (KIT), welches aus der Fusion der Universität Karlsruhe (TH) und dem Forschungszentrum Karlsruhe (FZK) hervorgegangen ist. Ich möchte all jenen danken, die während dieser Zeit zum Gelingen meiner Arbeit beigetragen haben.

Mein Dank gilt Herrn Prof. Dr. Vincent Heuveline, der diese Arbeit erst ermöglicht hat, sowie Herrn Jun.-Prof. Dr. Jan-Philipp Weiß und Frau PD Dr. Gudrun Thäter für die vielen wertvollen und hilfreichen Anregungen und die immerwährende Unterstützung in sämtlichen Angelegenheiten. Herr Prof. Dr. Alefeld danke ich sehr für die Unterstützung bei der Fertigstellung dieser Arbeit. Mein weiterer Dank gilt den Mitarbeitern des Institutes für die exzellente Atmosphäre und die hervorragende Zusammenarbeit.

Meiner Frau Julia und meinen beiden Kindern, Maja-Lina und Liana-Lotta, möchte ich für die seelische und moralische Unterstützung, die Geduld und die wunderbare Zeit, die ich mit Ihnen verbringen durfte, an dieser Stelle in besonderem Maße danken. Ich freue mich, dass Ihr da seit.

Herzlichen Dank Euch allen!

Björn Rocker

Karlsruhe im Dezember 2010

Preface

This dissertation originates from my work at the Institut für Angewandte und Numerische Mathematik 4 (NumHPC) and the Engineering Mathematics and Computing Lab (EMCL). Both institutions are part of the Karlsruhe Institute of Technology (KIT) which is the a merger of the Universität Karlsruhe (TH) and the Forschungszentrum Karlsruhe (FZK).

I want to thank all who were supporting my work during that time. I express my special thanks to Prof. Dr. Vincent Heuveline, who made this work possible in the first place. Further, I am very grateful to Jun.-Prof. Dr. Jan-Philipp Weiß, PD Dr. Gudrun Thäter and Prof. Dr. Alefeld for their many valuable suggestions and their support in all aspects related to my work. My thanks also goes to my colleagues for the excellent working atmosphere as well as rewarding collaborations.

In a special way, I want to honor wife Julia and my two children Maja-Lina and Liana-Lotta. I am grateful for their mental and moral support, for their patience and for the wonderful time I have spent with them.

Many thanks to all of you!

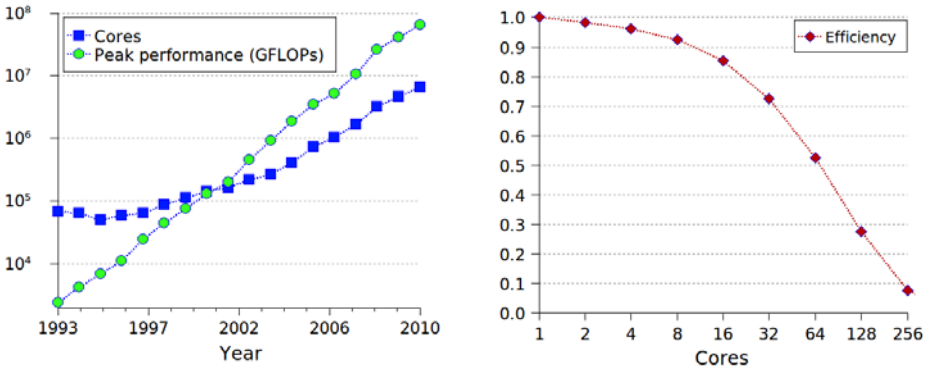
Björn Rocker

Karlsruhe, December 2010

Contents

Introduction

Users with academic, industrial or research background employ high performance computers to perform simulations, often addressing socially relevant problems in engineering, medicine and meteorology. This is the case for multiple reasons. Achieving cost-savings by replacing physical experiments, economizing time or avoiding risks for human beings and material are the most mentioned ones. In some areas, numerical simulation is the only way to gain information, since experiments are not possible (e.g. astronomy, medicine or climate research). As a result, better products can be designed, the quality of life can be improved and lives and real assets can be saved. All this is possible due to the massive computational power that is available nowadays and numerical simulation becomes more and more important in many areas.



(a) (b)

Figure 0.1. (a) Sum of cores and accumulated peak performance of all computers in the top500 list over time from 1993 till 2010. (b) Empiric efficiency[1] of many parallel solvers for sparse linear systems over the number of cores.

However, the new hardware capabilities and current developments for future computing architectures offer multiple challenges for their users and especially for the underlying numerical methods.

Todays hardware architectures have reached an enormous complexity which is visible from the number of cores of all computers in the Top500 [**top10**], a list of the 500 most powerful supercomputers in the world with respect to their absolute performance in the Linpack benchmark [**DLP03**], as displayed in Figure 0.1(a).

[1] Speedup (S) is defined as the quotient of the time (T_s) the fastest serial algorithm needs divided by the time (T_p) a parallel executed algorithm on p cores needs for solving the problem ($S := \frac{T_s}{T_p}$). In some cases the parallel algorithm executed on one core (T_1) is taken instead of the fastest serial algorithm (T_s) which can lead to very different results in some cases.
Efficiency (E) is defined as speedup divided by the number of used cores within the parallel execution ($E := \frac{S}{p}$).

The average size of clusters in the list is between four- and eight-thousand cores (see Figure 0.2), whereas some of the most powerful machines host several hundred thousand cores. Besides machines dedicated designed for high-performance computing, there are computing centers, mostly in the area of cloud-computing, hosting machines with millions of cores.

It can also be observed that an increasing number of machines are equipped with accelerators like graphic processing units (GPUs) or consist of other heterogeneous hardware components offering very special capabilities (e.g. Cell, FPGAs etc.). Such architectures offer very often the best performance per Watt-ratio according to the Green500 list [**gre10**] but are usually intricate to use. Although a lot of research is done to use the computational resources in an efficient way (by means of creating as many meaningful results as possible based on a given amount of resources), many solvers do not exploit the available performance (even for conventional CPU-based architectures) which is suggested in Figure 0.1(b). This disadvantageous behavior is especially the case when problems are solved that are fully coupled, which is e.g. often the case in computational fluid dynamics (CFD). When incompressible, Newtonian fluids are simulated, finite element (or related) approaches are very frequently employed, usually leading to very large, sparse linear systems that have to be solved. Since the solution of these linear systems is typically the most time consuming step in the solution process, research in this area is essential. "Hardware-aware" numerical methods is one emerging expression and connotes, that within the selection and / or development process of numerical methods for solving a problem, the characteristics of hardware technologies were assigned to play a key role.

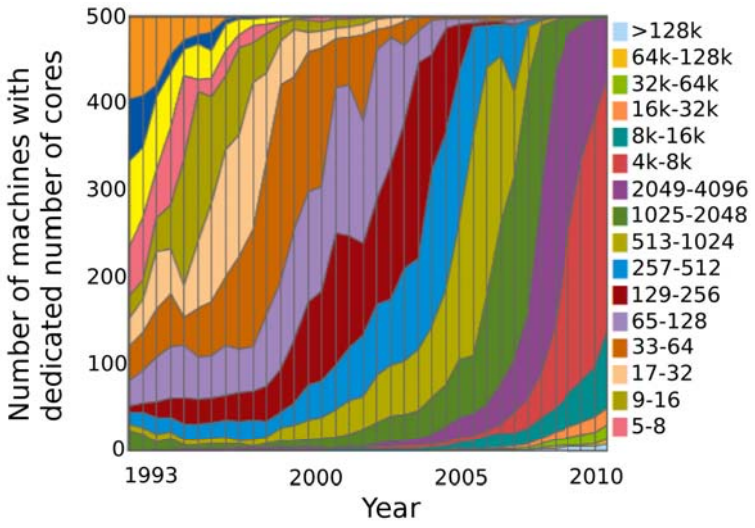


Figure 0.2. Number of cores of the computers in the top500 list from 1993 till 2010. Graphic taken from [**top10**].

In Figure 0.1(a) the accumulated theoretical peak performance of the worlds fastest computers on the Top500 list is shown in order to illustrate the enormous available computational performance. Today (2010) there are, in terms of theoretical peak performance, the first petascale architectures available, offering the possibility to perform $O(10^{15})$ floating point operations per second (FLOPS). Previous terascale architectures ($O(10^{13})$ FLOPS) are wide spread across the whole world. Due to the fact that the capabilities numerical simulations offer becomes evident and additional computing power is requested to consider more complex problems, researchers aim

at the development of the next generation of supercomputers offering exascale capabilities ($O(10^{18})$ FLOPS) (e.g. the Exascale Innovation Center in Jülich founded by IBM and the FZ Jülich or the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL)). Claimed goal is to reach exascale within the next decade. Beside the development of hardware components and software to operate an exascale computer, it has to be guaranteed that applications can benefit from the computational resources. This is only the case by enhancing (in close interaction with developments of hardware and software) the needed numerical methods and algorithms in such a way that they exhibit fault tolerance properties and scale well (meaning, that additional resources can be used in an efficient way).

Beside computing results, reproducibility of experiments is one basic request for scientific research. That rounding errors can have a significant impact on the accuracy of numerical results is known since decades and the situation gets worse on today's massive parallel and heterogeneous computers. It is essential to be aware of the effects that can occur due to the finite (and therefore potentially inexact) arithmetic on the computer and to know when techniques like interval arithmetics have to be employed in order to get verified results. Similar problems arise when hardware technologies shall be used which offer only a certain accuracy while a higher one is requested for the result. Even for such circumstances there exist solutions.

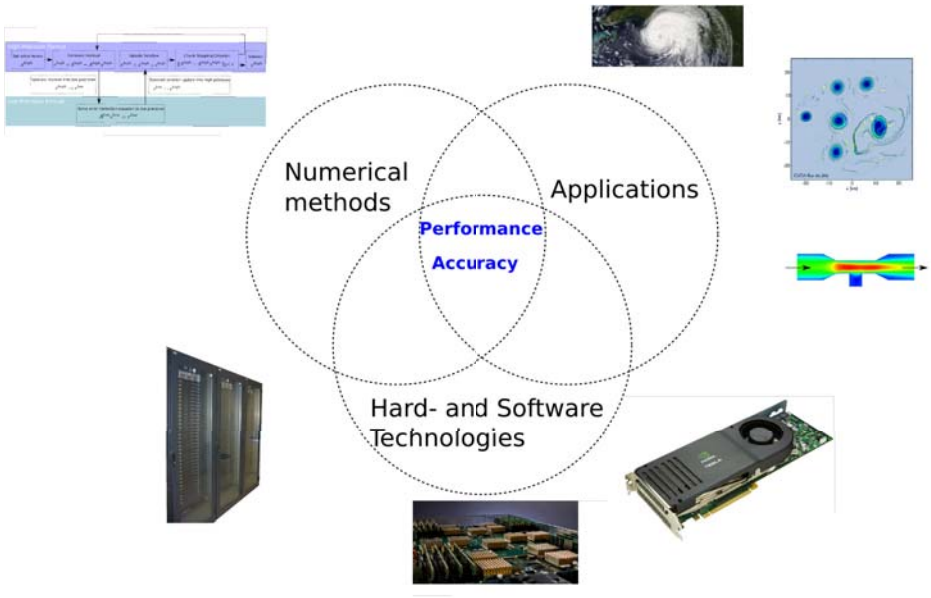


Figure 0.3. Focus of the thesis are the two topics accuracy and performance of solvers for large, sparse linear systems from a holistic point of view encompassing both, the numerical as well as the technological factors. Many applications can directly benefit from the presented results, as for instance the accelerated meteorological simulation demonstrates.

Within this thesis, the above described challenges of accuracy, heterogeneity and parallelization are addressed with respect to iterative solvers for linear systems of equations. The existence and effectiveness of numerical procedures exploiting capabilities of new hardware technologies is shown. Parallelization strategies for the method of conjugate gradients (CG) and the generalized minimum residual (GMRES) method are presented with respect to special hard- and software. Effects of

the data distribution on accuracy and performance are analyzed enabling optimization with respect to accuracy, respectively performance, to achieve a higher quality of the results. Based on the presented findings many numerical simulations can be accelerated and more complex problems can be solved.

Goal of the thesis is to give an holistic view on numerical mathematics in the context of modern hardware technologies with focus on iterative solvers for sparse linear systems in terms of performance and accuracy. For solving a problem efficiently, the need for a deep understanding of both the numerical methods as well as modern hardware technologies can not be overemphasized.

In many applications, processes are modeled by partial differential equations and finite element methods are frequently used to discretize such problems. Usually large, sparse linear problems have to be solved and the method of conjugate gradients (CG) and the generalized minimum residual (GMRES) method are frequently employed. Both methods, as well as some basic preconditioning techniques, are explained in **Chapter 1**.

Chapter 2 gives an overview of modern hardware technologies. Compendiously explained are multiprocessor systems, virtual memory arrangements, graphic processing units (GPUs) and field programmable gate arrays (FPGAs). Also presented are the programming paradigms (or application programming interfaces (APIs)) MPI, OpenMP, CUDA, OpenCL and PGAS languages.

Multiprecision, respectively mixedprecision, methods are discussed in **Chapter 3** in order to take advantage of the performance in different floating point formats offered by hardware but to concurrently ensure a certain quality of the result. A major class of such methods can be represented as iterative refinement methods, which is the focus of the chapter. Convergence of mixed precision iterative refinement solvers is analyzed and numerical experiments are presented to validate the theoretical findings. For the special case of elliptic operators, analytical bounds are derived ensuring that the method error stays in the worst case in the same order of magnitude as the discretization error.

In the following **Chapter 4**, strategies for highly efficient parallel CG and GMRES implementations on a CPU cluster and a GPU based architectures are presented when MPI, OpenMP or CUDA is used. Based on numerical experiments with multiple test-cases and a comparison with a state of the art software library, the quality is validated. Thereby, a strong emphasis is set to applications based on partial differential equations in the area of fluid dynamics. Lastly, the impact of preconditioning is also addressed.

In order to get information about the capabilities of hardware, prototypical evaluations are shown in **Chapter 5**. This is done for CPU-, GPU- as well as for FPGA-based architectures. Elementary kernels as well as full solvers for practice-oriented applications have been performed. All architectures are considered separately as well as in an inter-architectural comparison.

Rounding errors in combination with parallelization are addressed in **Chapter 6**. Basic theory of rounding error propagation in elementary operations as well as the two dimensional block-cyclic data distribution is presented. The theoretical findings are validated based on numerical experiments, whereby verified computing in terms of interval arithmetics is applied to obtain reliable results.

In the context of the previous findings, **Chapter 7** presents results how different data distributions can effect the achievable accuracy of a preconditioned CG solver.

In **Chapter 8**, the acceleration of an existing simulation for tropical cyclones, used by researchers from the meteorology, is demonstrated. After an short overview about the meteorological model and the solving process, results for the implementation for a GPU are presented.

Finally, **Chapter 9** summarizes the results of the thesis and gives an outlook.

CHAPTER 1

Selected Solvers for Linear Systems of Equations

Within this chapter, the method of conjugate gradients (CG), as well as the generalized minimal residual method (GMRES) are explained from the theoretical point of view. Preconditioning techniques for linear solvers are also shown and a short summary of stopping criterias close the chapter. Parts of this chapter can also be found in [**Gal10**] and partially follow the path of [**Fac00**], [**SK06**] and [**Saa03**].

1.1. Projection Methods

The basic idea of projection methods for solving a linear System $Ax = b$, with $A \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$, is to search an approximate solution x_k in a certain subspace $\mathcal{V}$ of $\mathbb{R}^n$. The chosen approximation is optimal in $\mathcal{V}$ if the error, defined as $x^* - x_k$ with the correct solution x^*, stays orthogonal on $\mathcal{V}$. Since one cannot compute the error without the solution x^* the residual $r_m = Ax_m - b$ is chosen to be orthogonal to a second certain subspace $\mathcal{W}$. In general there are *orthogonal* and *skew* (or *oblique*) projection methods. The subspaces $\mathcal{V}$ and $\mathcal{W}$ are the same for orthogonal projection methods and different for skew projection methods. Projection methods are defined as follows:

Definition 1.1. A projection method for solving $Ax = b$ searches a approximate solution $x_m \in \mathcal{V}_m$ with

$$r_m = b - Ax_m \perp \mathcal{W}_m, \tag{1.1}$$

where $\mathcal{V}_m$ and $\mathcal{W}_m$ are m-dimensional subspaces of $\mathbb{R}^n$.

For a orthogonal projection method ($\mathcal{V}$=$\mathcal{W}$) (1.1) is called a *Galerkin condition*. For a skew projection method ($\mathcal{V} \neq \mathcal{W}$) (1.1) is called *Petrov-Galerkin condition* [**Fac00, Saa03**].

If an initial guess x_0 is considered, the approximate solution is searched in the affine subspace $x_0 + \mathcal{V}$. By assuming the availability of a basis for $\mathcal{V}$ and a basis for $\mathcal{W}$ the matrices V and W can be defined whose columns are the basis vectors of $\mathcal{V}$ and $\mathcal{W}$. Now it is possible to find a representation for the approximate solution x_k in $x_0 + \mathcal{V}$ as follows:

$$x_k = x_0 + Vy. \tag{1.2}$$

With the orthogonality condition (1.1) one gets for y

$$\begin{aligned} & W^T r_k = 0 \\ \Leftrightarrow\ & W^T(b - A(x_0 + Vy)) = 0 \\ \Leftrightarrow\ & W^T(r_0 - AVy) = 0 \\ \Leftrightarrow\ & W^T AVy = W^T r_0. \end{aligned} \tag{1.3}$$

Combining (1.2) and (1.3) one obtains

$$x_k = x_0 + V(W^T AV)^{-1} W^T r_0.$$

Algorithm 1 General Projection Method

1: **repeat**
2: Select $\mathcal{V}$ and $\mathcal{W}$
3: Compute bases and build matrices V and W
4: $r \leftarrow b - Ax$
5: $y \leftarrow (W^T AV)^{-1} W^T r$
6: $x \leftarrow x + Vy$
7: **until convergence**

This leads to a general form of a projection method as shown in Algorithm 1. Note that algorithm 1 is meaningful if $W^T AV$ is nonsingular. There are two important cases for which this constraint holds. First, if A is positive definite and $\mathcal{V} = \mathcal{W}$ which leads for example to the method of Conjugate Gradients. Secondly, if A is nonsingular and $\mathcal{V} = A\mathcal{W}$ which leads to the GMRES method. A proof to this two statements as well as different theoretical results for projection methods can be found in [**Saa03**].

1.2. Krylov Subspaces

A general Krylov subspace with respect to a matrix A is defined as follows:

Definition 1.2.

$$\mathcal{K}_m(A, v) := \text{span}\{v, Av, A^2v, ..., A^{m-1}v\}, \qquad m = 1, 2, ...$$

is called a Krylov subspace.

It yields $\dim(\mathcal{K}_m) = \min(m, \text{grade}(v))$, where $\text{grade}(v)$ is the degree of the minimal polynomial of the vector v with respect to A.

For projection methods onto Krylov subspaces and Krylov type iterative solvers, it is important to compute an orthogonal basis of the related Krylov subspaces. For general matrices this can be done with the Arnoldi algorithm (see Section 1.2.1) whereas the Lanczos algorithm can be used in the case of a symmetric and positive definite matrix. Figure 1.1 shows a survey of common Krylov type solvers for linear systems, the main developers and the year of publication, the algorithm for computing the orthogonal basis and a main characteristic of the algorithm.

1.2.1. Arnoldi Method. The Arnoldi Method can be used to compute an orthogonal basis of a Krylov subspace $\mathcal{K}_m$. In the basic version Gram-Schmidt conjugation is used to orthonormalize basis vectors. The vectors $v_1, v_2, ..., v_m$ pro-

Algorithm 2 Arnoldi Method using Gram-Schmidt

1: Choose a vector v_1 with $||v_1||_2$=1
2: **for** $j = 1, 2, ..., m$ **do**
3: **for** $i = 1, .., j$ **do**
4: $h_{ij} = \langle Av_j, v_i \rangle$
5: $w_j := Av_j - \sum_{i=1}^{j} h_{ij} v_i$
6: $h_{j+1,j} = ||w_j||_2$.
7: If $h_{j+1,j} = 0$ stop
8: $v_{j+1} = \frac{1}{h_{j+1,j}} w_j$

duced by Algorithm 2 form an orthonormal basis of $\mathcal{K}_m(A, v_1)$. These vectors build

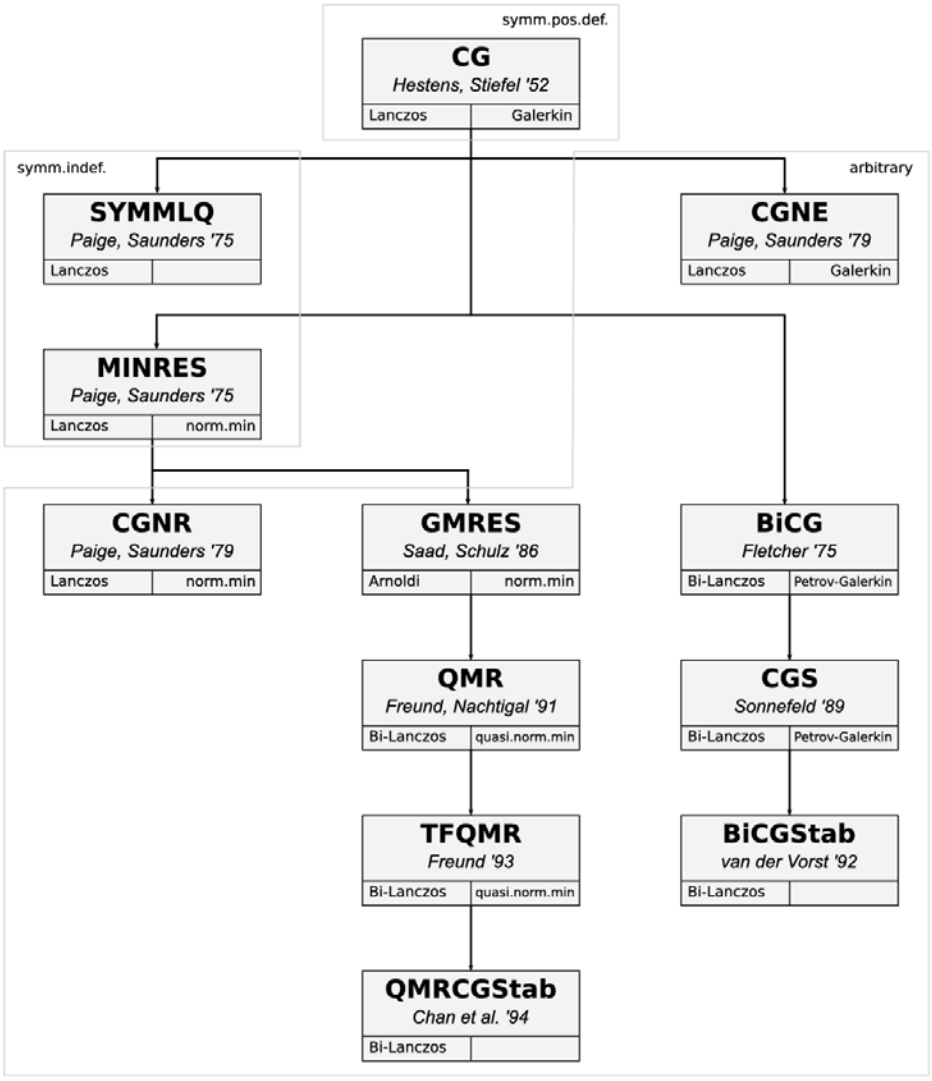


Figure 1.1. Historical development of Krylov type solvers, taken from [**Fac00**]. Mentioned are the names of the solvers, their developers, when they have been published, the algorithm for building the Krylov subspaces and the principal method they are based on.

a matrix

$$V_m := (v_1|v_2|...|v_m) \in \mathbb{R}^{n\times m} \qquad \text{with } V_m^T V_m = I_m.$$

The constants produced in line 4 form a $(m+1)\times m$ Hessenberg matrix H_m, which is defined as

$$H_m := \begin{pmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1m} \\ h_{12} & h_{22} & h_{23} & \cdots & h_{2m} \\ 0 & h_{32} & h_{33} & \cdots & h_{3m} \\ 0 & 0 & h_{43} & \cdots & h_{4m} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & h_{m+1,m} \end{pmatrix}.$$

With line 6 in Algorithm 2 it follows

$$Av_m = \sum_{i=1}^{m+1} h_{im} v_i, \qquad m = 1, 2, 3, ...$$

and therefore one obtains

$$AV_m = V_{m+1} H_m, \qquad m = 1, 2, 3, ... \ . \tag{1.4}$$

Furthermore it yields

$$AV_m = V_m \bar{H}_m + w_m e_m^T \tag{1.5}$$

$$V_m^T A V_m = \bar{H}_m \tag{1.6}$$

where $\bar{H}_m$ is obtained by deleting the last row of H_m. These properties are important for the GMRES algorithm.

In practice the orthogonality of these vectors is destroyed by rounding errors. To minimize this effect, a modified Gram-Schmidt method can be used or the Householder algorithm for building the orthonormal basis. More information can be found in [**Fac00**], [**SK06**] and [**Saa03**].

Algorithm 3 Arnoldi Method using modified Gram-Schmidt

1: Choose a vector v_1 with $||v_1||_2$=1
2: **for** $j = 1, 2, ..., m$ **do**
3: $w_j := Av_j$
4: **for** $i = 1, .., j$ **do**
5: $h_{ij} = \langle w_j, v_i \rangle$
6: $w_j := w_j - h_{ij} v_i$
7: $h_{j+1,j} = ||w_j||_2$. If $h_{j+1,j} = 0$ stop
8: $v_{j+1} = \frac{1}{h_{j+1,j}} w_j$

1.2.2. Lanczos Algorithm. In the special case of a symmetric matrix A, the Arnoldi algorithm can be reduced to the Lanczos Algorithm. It follows from (1.6) that for a symmetric matrix A the matrix $\bar{H}_m$ becomes also symmetric.

$$\bar{H}_m^T = (V_m^T A V_m)^T = V_m^T (V_m^T A)^T = V_m^T A^T V_m = V_m^T A V_m = \bar{H}_m.$$

Due to the fact that $\bar{H}_m$ is a Hessenberg matrix, it must be tridiagonal. Therefore line 6 in Algorithm 2 can be simplified to a three-term recurrence. More precisely, a new basis vector need not to be orthogonalized against all previous vectors, only against the last three vectors. With $\alpha_j = h_{jj}$ and $\beta_j = h_{j-1,j}$ the matrix $\bar{H}_m$ reduces to

$$T_m := \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix}.$$

Algorithm 4 describes the basic Lanczos Method. For more information see [**Saa03, Fac00**].

Algorithm 4 Lanczos Algorithm

1: Choose a vector v_1 with $||v_1||_2$=1, $\beta_1 = 0$,$v_0 = 0$
2: **for** $j = 1, 2, ..., m$ **do**
3: $w_j := Av_j - \beta_j v_{j-1}$
4: $\alpha_j = \langle w_j, v_j \rangle$
5: $w_j := w_j - \alpha_j v_j$
6: $\beta_{j+1} = ||w_j||_2$.
7: If $\beta_{j+1} = 0$ stop
8: $v_{j+1} = \frac{1}{\beta_{j+1}} w_j$

1.3. Conjugate Gradients Method

In this section the method of Conjugate Gradients is described. This method is often used for linear systems with a positive definite and symmetric matrix A. In this section it is assumed that the matrix A has these properties.

There are various derivations for the CG-method, e.g. as a projection Method onto the Krylov subspace $\mathcal{K}_m(A, r_0)$. In the following the CG is derived from improvements of the Steepest Decent method via the method of Conjugate Directions.

Solving the linear system $Ax = b$ with a symmetric and positive definite matrix A is equivalent to minimize

$$f(x) = \frac{1}{2} x^T Ax - b^T x. \tag{1.7}$$

Note that $f'(x) = Ax - b$. The error is defined as $e_j = x_j - x^*$. It yields

$$\begin{aligned} & x^* = A^{-1}b \\ \Leftrightarrow\ & x_j - x^* = x_j - A^{-1}b \\ \Leftrightarrow\ & e_j = A^{-1}(Ax_j - b) \\ \Leftrightarrow\ & e_j = -A^{-1}r_j. \end{aligned}$$

$r_j = b - Ax_j$ is called the residual and note that $r_j = -f'(x_j)$.

The Method of **Steepest Descent** (or Gradient Descent) provides a sequence $x_1, x_2, ...$, which converges to the solution x^* ($f(x^*) = \min f(x)$). Starting from an initial guess, the method takes in every iteration cycle a step towards the solution in the negative direction of the gradient of f at the current point,

$$x_{j+1} = x_j - \alpha_j \nabla f(x_j) = x_j - \alpha_j f'(x_j) = x_j + \alpha_j r_j.$$

In every step the direction is chosen in which f decreases most quickly. α_j determines the increment in this direction. The optimal α_j is chosen to minimize f along $x_j - tr_j$. This is called *line search*. To determine α_j the directional derivative $\frac{d}{d\alpha_j} f(x_{j+1})$ is set equal to zero. With

$$\frac{d}{d\alpha_j} f(x_{j+1}) = f'(x_{j+1})^T \frac{d}{d\alpha_j} x_{j+1} = f'(x_{j+1})^T r_j \tag{1.8}$$

and $f'(x_{j+1}) = -r_{j+1}$ one gets

$$\begin{aligned}
\frac{d}{d\alpha_j} f(x_{j+1}) &= 0 \\
\Leftrightarrow r_{j+1}^T r_j &= 0 \\
\Leftrightarrow (b - Ax_{j+1})^T r_j &= 0 \\
\Leftrightarrow (b - A(x_j + \alpha_j r_j))^T r_j &= 0 \\
\Leftrightarrow (b - Ax_j)^T r_j &= \alpha_j (Ar_j)^T r_j \\
\Leftrightarrow r_j^T r_j &= \alpha_j r_j^T (Ar_j) \\
\Leftrightarrow \alpha_j &= \frac{r_j^T r_j}{r_j^T A r_j}.
\end{aligned}$$

Additionally the orthogonality between r_{j+1} and r_j can be seen. Altogether the following algorithm can be derived:

Algorithm 5 Steepest Descent

1: $r_i = b - Ax_i$
2: $\alpha_i = \frac{\langle r_i, r_i \rangle}{\langle r_i, Ar_i \rangle}$
3: $x_{i+1} = x_i + \alpha_i r_i$

For more details see [**Bra07**] and [**SB05**].

One problem of the method of Steepest Descent is, that the algorithm takes steps in the same direction as earlier steps did, leading potentially to poor convergence. The idea to improve this, is to use orthogonal search directions. This leads to the method of Conjugate Directions.

The Method of **Conjugate Directions** uses a set of orthogonal directions $\{p_0, p_1, ..., p_{n-1}\}$ and takes only one step in every search direction

$$x_{j+1} = x_j + \alpha_j p_j. \tag{1.9}$$

To achieve this the error $e_{j+1} = x_{j+1} - x^*$ should be orthogonal to p_j. This is used to determine α_j

$$\begin{aligned}
p_j^T e_{j+1} &= 0 \\
\Leftrightarrow p_j^T (e_j + \alpha_j p_j) &= 0 \\
\Leftrightarrow \alpha_j &= -\frac{p_j^T e_j}{p_j^T p_j}.
\end{aligned}$$

Here arises a problem. It is not possible to compute the error e_j without knowing x^*. The solution is to use A-orthogonal (or conjugated) search directions, this means

$$\langle p_i, p_j \rangle_A = p_i^T A p_j = 0 \ , \ p_i \neq p_j.$$

Now e_{j+1} shall be A-orthogonal to p_j. For determining α_j one has to minimize f along $x_j - tp_j$. With (1.8) and (1.9) it follows that

$$\begin{aligned}
\frac{d}{d\alpha_j} f(x_{j+1}) = f'(x_{j+1})^T p_j &= 0 \\
\Leftrightarrow -r_{j+1}^T p_j &= 0 \\
\Leftrightarrow p_j^T A e_{j+1} &= 0 \\
\Leftrightarrow p_j^T A(e_j + \alpha_j p_j) &= 0 \\
\Leftrightarrow \alpha_j &= -\frac{p_j^T A e_j}{p_j^T A p_j} = \frac{p_j^T r_j}{p_j^T A p_j}.
\end{aligned}$$

Together with a set of A-conjugate vectors $p_0, ..., p_{n-1}$ the algorithm for the method of conjugate directions can be obtained as shown in Algorithm 6.

Algorithm 6 Conjugate Directions

1: select a set of A-conjugate directions $p_0, ..., p_{n-1}$
2: **repeat**
3: $r_i = b - Ax_i$
4: $\alpha_j = \frac{p_j^T r_j}{p_j^T A p_j}$
5: $x_{j+1} = x_j + \alpha_j p_j$
6: **until** convergence

The matrix-vector product for determining the residual r_j can be replaced by a recursion. With $e_{j+1} = e_j + \alpha_j p_j$ one gets

$$\begin{aligned} r_{j+1} &= -Ae_{j+1} \\ &= -A(e_j + \alpha_j p_j) \\ &= r_j - \alpha_j A p_j. \end{aligned} \tag{1.10}$$

Note that this method terminates with exact arithmetic after n steps for a $n \times n$-matrix, see [**ALO02**]. For building the set of A-orthogonal search directions, one can use the *Gram-Schmidt process*. Here the problem arises, that all old directions are needed to build a new one leading to a high complexity for building the set of conjugate directions.

The method of Conjugate Directions has an special property. In every step it minimizes the error e_j regarding to the *energy norm* $||e||_A = (e^T A e)^{\frac{1}{2}}$ in $e_0 + \mathcal{D}_\mathrm{j}$. $\mathcal{D}_\mathrm{j}$ is defined as the subspace span$\{p_0, ...p_{j-1}\}$. To show this the error can be written as

$$e_0 = x_0 - x^* = \sum_{i=0}^{n-1} \delta_i p_i \tag{1.11}$$

with the set of orthogonal directions $p_0, p_1, ..., p_{n-1}$. Now one can write e_j as

$$\begin{aligned} e_j = x_j - x^* &= e_0 + \sum_{i=0}^{j-1} \alpha_i p_i \\ &= \sum_{i=0}^{n-1} \delta_i p_i - \sum_{i=0}^{j-1} \delta_i p_i \\ &= \sum_{i=j}^{n-1} \delta_i p_i \end{aligned} \tag{1.12}$$

In the second line the relation $\delta_i = -\alpha_i$ (see [**She94**]) is used. It follows from (1.12) that e_j is in $e_0 + \mathcal{D}_\mathrm{j}$. Consider the energy norm of e_j

$$||e_j||_A^2 = \sum_{i=j}^{n-1} \sum_{k=j}^{n-1} \delta_k \delta_i p_k^T A p_i = \sum_{i=j}^{n-1} \delta_i^2 p_i^T A p_i. \tag{1.13}$$

Here it can be seen that e_j has the minimum energy norm in $e_0 + \mathcal{D}_\mathrm{j}$, because any other vector of this subspace has at least the same terms in its expansion (see (1.11) and (1.12)).

The Method of **Conjugate Gradients** is an improvement of the methods of Steepest Descent and Conjugate Directions where the disadvantage in building the search directions disappears. By conjugation of the residuals the search directions are constructed and it is no longer needed to store the old search vectors (see [**Saa03**]).

Following relations are needed to proceed. First the orthogonality of r_j and span$\{p_0, ..., p_{j-1}\}$ is shown:

$$p_i^T A e_j = \sum_{j=i}^{n-1} \delta_j p_i^T A p_j$$
$$p_i^T r_j = 0, i < j.$$

Furthermore r_j is orthogonal to all previous residuals. To show this the inner product of equation (1.15) and r_i is considered.

$$p_j^T r_i = r_j^T r_i + \sum_{k=0}^{j-1} \beta_{jk} p_k^T r_i \qquad 0 = r_j^T r_i, j < i. \tag{1.14}$$

Obviously $r_j^T r_i = 0$ holds for $i \neq j$ and with the first equation above one gets $p_i^T r_i = r_i^T r_i$. Now the goal is to build the conjugate directions from the residuals. Using the Gram-Schmidt conjugation for building these directions, it is set $p_0 = r_0$ and for $j > 0$

$$p_j = r_j + \sum_{k=0}^{j-1} \beta_{jk} p_k. \tag{1.15}$$

To estimate β_{jk}, (1.15) is transformed into

$$\begin{aligned} p_j^T A p_i &= r_j^T A p_i + \sum_{k=0}^{j-1} \beta_{jk} p_k^T A p_i \\ 0 &= r_j^T A p_i + \beta_{ji} p_i^T A p_i \ , j > i \\ \beta_{ji} &= -\frac{r_j^T A p_i}{p_i^T A p_i}. \end{aligned} \tag{1.16}$$

To simplify this expression the inner product of equation (1.10) and r_i is taken:

$$r_i^T r_{j+1} = r_i^T r_j - \alpha_j r_i^T A p_j \tag{1.17}$$
$$\alpha_j r_i^T A p_j = r_i^T r_j - r_i^T r_{j+1} \tag{1.18}$$
$$r_i^T A p_j = \begin{cases} \frac{1}{\alpha_i} r_i^T r_i, & i = j, \\ -\frac{1}{\alpha_{i-1}} r_i^T r_i, & i = j+1, \\ 0, & \text{otherwise.} \end{cases} \tag{1.19}$$
$$\beta_{ij} = \begin{cases} \frac{1}{\alpha_{i-1}} \frac{r_i^T r_i}{p_{i-1}^T A p_{i-1}}, & i = j+1, \\ 0, & i > j+1. \end{cases} \tag{1.20}$$

In (1.19) the relation (1.14) is used. Now it can be seen that almost all β_{ij} in the Gram-Schmidt process disappear. More precisely, only the last direction is needed to build the next conjugate direction. This makes the Conjugate Gradient Method very useful. With a trick the matrix-vector product in the representation of β (1.20) can be avoided. Using the representation of α in algorithm 6, which does not change for Conjugate Gradient, and the identity (1.14) one gets

$$\beta_i = \frac{r_i^T r_i}{p_{i-1}^T r_{i-1}} = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}}.$$

Identity (1.14) provides

$$\alpha_i = \frac{p_i^T r_i}{p_i^T A p_i} = \frac{r_i^T r_i}{p_i^T A p_i}.$$

Algorithm 7 shows the method of Conjugate Gradients.

Algorithm 7 Conjugate Gradients

1: $r_0 = b - Ax_0$, $p_0 = r_0$, A spd
2: **for** $i = 0, 1, 2, ...$ **do**
3: $\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i}$
4: $x_{i+1} = x_i + \alpha_i p_i$
5: $r_{i+1} = r_i - \alpha_i A p_i$
6: $\beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
7: $p_{i+1} = r_{i+1} + \beta_i p_i$

Note that the search directions are built by the residuals. Thus it follows

$$\mathcal{D}_\mathrm{j} = \mathrm{span}\{p_0, ...p_{j-1}\} = \mathrm{span}\{r_0, ...r_{j-1}\}.$$

In addition, the residuals are built by the old residuals and Ap_{j-1}. Therefore it holds

$$\mathcal{D}_\mathrm{j} = \mathrm{span}\{r_0, Ar_0, ...A^{j-1}r_0\} = \mathcal{K}_{j-1}(A, r_0).$$

Now one can see that the method of Conjugate Gradients is also an orthogonal projection method, where $\mathcal{V}_j = \mathcal{K}_j$. Relation (1.14) ensures the orthogonality of the residual and $\mathcal{W}$. For a derivation of CG as a projection method see [**Saa03**].

Note that the method of Conjugate Gradients is only a special version of Conjugate Directions. It uses the residuals to build up the search directions. Therefore Conjugate Gradients has the same error minimizing property as Conjugate Directions. The error e_j is minimized regarding to the Krylov subspace $\mathcal{K}_j(A, r_0)$. Additionally, the error satisfies

$$||e_j||_A \overset{1.13}{=} \sum_{i=j}^{n-1} \delta_i^2 p_i^T A p_i > \sum_{i=j+1}^{n-1} \delta_i^2 p_i^T A p_i = ||e_{j+1}||_A.$$

Note that $\delta_i^2 p_i^T A p_i > 0$ with A positive definite.

There are one matrix-vector product, three vector updates and two dot-products per iteration cycle. In general the matrix-vector product for computing Ap_j needs n^2 floating-point multiplications and $n^2 - n$ summations, leading to a asymptotic complexity of $O(n^2)$. The complexity for the vector updates is $O(n)$, because n multiplications and n summations for each update are needed. The inner product has also a complexity of $O(n)$. Hence the total complexity per iteration step is dominated by the matrix-vector product. If sparse matrices are used and only nonzero entries are saved (see Chapter 4.1) the complexity decreases. Supposing a matrix having nnz nonzero entries and $nnz << n^2$. Now, nnz floating-point multiplications are needed and at most $nnz - 1$ summations. The total complexity is $O(nnz)$ compared to $O(n^2)$ in the dense case.

As mentioned above CG is actually a direct method, which needs n steps to estimate the solution. Whereas for concerning convergence of CG is the fact that in many applications very large systems are handled and it is very expensive or impossible to perform n iterations.

First, note that

$$r_j = b - Ax_j = A(A^{-1}b - x_j) = A(x^* - x_j) = -Ae_j. \tag{1.21}$$

Due to the error optimality of CG it yields

$$e_k = x_k - x^* = \min\{z - x^*\}$$

with $z \in x_0 + \mathcal{K}_k(A, r_0)$, see also (1.12). Using the Krylov subspace and (1.21) one gets

$$\begin{aligned} e_k &= e_0 + c_1 A e_0 + c_2 A^2 e_0 + ... + c_k A^k e_0 \\ &= (I + c_1 A + c_2 A^2 + ... + c_k A^k) e_0 \\ &= P_k(A) e_0 \end{aligned}$$

with a polynomial $P_k(t)$ of degree $\leq k$ satisfying $P_k(0) = 1$. Using the error optimality again, it can be seen that

$$||e_k||_A = \min_{P_k(t)} ||P_k(A) e_0||_A. \tag{1.22}$$

For the positive definite matrix A exists a set of n real and positive eigenvalues $0 < \lambda_1 \leq ... \leq \lambda_n$ and corresponding orthonormal eigenvectors $z_1, ..., z_n$. Furthermore, there exists a unique exposition of

$$e_0 = c_1 z_1 + c_2 z_2 + ... + c_n z_n.$$

Now $||e_k||_A$ and $||e_0||_A$ are considered:

$$||e_0||_A^2 = \langle \sum_{i=1}^{n} c_i z_i, \sum_{j=1}^{n} c_j \lambda_j z_j \rangle = \sum_{i=1}^{n} c_i^2 \lambda_i. \tag{1.23}$$

$$\begin{aligned} ||e_k||_A^2 &= ||P_k(A) e_0||_A^2 = \langle P_k(A) e_0, A P_k(A) e_0 \rangle \\ &= \langle \sum_{i=1}^{n} c_i P_k(\lambda_i) z_i, \sum_{j=1}^{n} c_j \lambda_j P_k(\lambda_j) z_j \rangle \\ &= \sum_{i=1}^{n} c_i^2 \lambda_i P_k^2(\lambda_i) \leq [\max_j \{P_k(\lambda_j)\}^2] \cdot ||e_0||_A^2. \end{aligned} \tag{1.24}$$

Combining (1.22) and (1.24) one obtains

$$\frac{||e_k||_A}{||e_0||_A} \leq \min_{P_k(t)} \{ \max_{\lambda \in [\lambda_1, \lambda_n]} |P_k(\lambda)| \}. \tag{1.25}$$

To proceed the Tschebyscheff polynomials and a relating theorem are needed. First, the Tschebyscheff polynomials are defined:

Definition 1.3. The Tschebyscheff polynomials are defined as

$$\begin{aligned} &T_0(x) := 1, T_1(x) := x, \\ &T_{n+1} := 2x T_n(x) - T_{n-1}(x). \end{aligned}$$

There is an interesting property of these polynomials:

$$\begin{aligned} &T_n(cos\theta) = cos(n\theta) \\ &T_n(x) = \cos(n\theta) = \frac{1}{2} \left[(x + \sqrt{x^2 - 1})^n + (x + \sqrt{x^2 - 1})^{-n} \right], x = \cos\theta. \end{aligned} \tag{1.26}$$

For more details see [**SK06**]. Finally the following theorem is needed:

Theorem 1.4. *For all polynomials $P_n(x)$ in $[-1, 1]$ with degree $n \geq 1$ and leading coefficient equal to one it yields*

$$\min_{P_n(X)} \{ \max_{x \in [-1,1]} |P_n(x)| \} = \max_{x \in [-1,1]} |\frac{1}{2^{n-1}} T_n(x)| = \frac{1}{2^{n-1}}.$$

For a proof and more details see [**SK06**]. To use this theorem with (1.25), the interval $[\lambda_1, \lambda_n]$ is transformed to $[-1, 1]$ by substituting x with $(2\lambda - \lambda_1 - \lambda_n)/(\lambda_n - \lambda_1)$. Thus

$$P_k(\lambda) := T_k\left(\frac{2\lambda - \lambda_1 - \lambda_n}{\lambda_n - \lambda_1}\right) / T_k\left(\frac{\lambda_1 + \lambda_n}{\lambda_1 - \lambda_n}\right)$$

has the lowest supremum norm in $[\lambda_1, \lambda_n]$, degree k and satisfies $P_k(0) = 1$. Furthermore is yields

$$\max_{\lambda \in [\lambda_1, \lambda_n]} |P_k(\lambda)| = 1/|T_k\left(\frac{\lambda_1 + \lambda_n}{\lambda_1 - \lambda_n}\right)|.$$

Defining

$$x := -\frac{\lambda_1 + \lambda_n}{\lambda_1 - \lambda_n} = \frac{\lambda_n/\lambda_1 + 1}{\lambda_n/\lambda_1 - 1} = \frac{\kappa(A) + 1}{\kappa(A) - 1} > 1$$

and using (1.26) one gets

$$T_k\left(\frac{\kappa + 1}{\kappa - 1}\right) = \frac{1}{2}\left[\left(\frac{\sqrt{\kappa} + 1}{\sqrt{\kappa} - 1}\right)^k + \left(\frac{\sqrt{\kappa} + 1}{\sqrt{\kappa} - 1}\right)^{-k}\right] \geq \frac{1}{2}\left(\frac{\sqrt{\kappa} + 1}{\sqrt{\kappa} - 1}\right)^k. \tag{1.27}$$

Finally, as a result of (1.25) and (1.27), one gets an upper bound for the error in step k in relation to the initial error in step 0.

$$\frac{||e_k||_A}{||e_0||_A} \leq 2\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k. \tag{1.28}$$

It is obvious that the convergence mainly depends on the condition number κ of A.

Equation (1.28) defines an upper bound for the number k of necessary iteration steps to achieve $||e_k||_A \leq \epsilon ||e_0||_A$:

$$k \leq \frac{1}{2}\sqrt{\kappa}\ln(\frac{2}{\epsilon}) + 1.$$

Now the goal is to increase the convergence speed of the CG method. Therefore the linear system $Ax = b$ is transformed to an equivalent system $\tilde{A}\tilde{x} = \tilde{b}$ with $\kappa(\tilde{A}) < \kappa(A)$, c.f. section 1.5. Symmetric preconditioning is used for this transformation to preserve the properties of A (see section 1.5). Hence

$$\begin{aligned} C^{-1}AC^{-T}\tilde{x} &= C^{-1}b, \\ \tilde{x} &= C^T x \\ M &= CC^T. \end{aligned}$$

The algorithm of Conjugate Gradients for this system is as shown in Algorithm 8. Computing C can be avoided by setting $\tilde{r}_i = C^{-1}$, $\tilde{p}_i = C^T p_i$ and using $\tilde{x}_i =$

Algorithm 8 Preconditioned Conjugate Gradients

1: $\tilde{p}_0 = -\tilde{r}_0 = C^{-1}b - C^{-1}AC^{-T}\tilde{x}_0$, A spd
2: **for** $i = 0, 1, 2, ...$ **do**
3: $\quad \alpha_i = \frac{\tilde{r}_i^T \tilde{r}_i}{\tilde{p}_i^T C^{-1}AC^{-T}\tilde{p}_i}$
4: $\quad \tilde{x}_{i+1} = \tilde{x}_i + \alpha_i \tilde{p}_i$
5: $\quad \tilde{r}_{i+1} = \tilde{r}_i + \alpha_i C^{-1}AC^{-T}\tilde{p}_i$
6: $\quad \beta_i = \frac{\tilde{r}_{i+1}^T \tilde{r}_{i+1}}{\tilde{r}_i^T \tilde{r}_i}$
7: $\quad \tilde{p}_{i+1} = -\tilde{r}_{i+1} + \beta_i \tilde{p}_i$

$C^T x_i$. Additionally, a vector h is used to minimize matrix-vector products with the

preconditioner M^{-1}. Altogether the optimized preconditioned CG-method can be found in Algorithm 9. Note that the search directions in Algorithm 9 are different

Algorithm 9 Optimized Preconditioned Conjugate Gradients (PCG)

1: $r_0 = b - Ax_0$, $h_0 = M^{-1}r_0$, $p_0 = h_0$, A spd, initial guess x_0
2: **for** $i = 0, 1, 2, ...$ **do**
3: $\quad \alpha_i = \frac{r_i^T h_i}{p_i^T A p_i}$
4: $\quad x_{i+1} = x_i + \alpha_i p_i$
5: $\quad r_{i+1} = r_i - \alpha_i A p_i$
6: $\quad h_{i+1} = M^{-1} r_{i+1}$
7: $\quad \beta_i = \frac{r_{i+1}^T h_{i+1}}{r_i^T h_i}$
8: $\quad p_{i+1} = h_{i+1} + \beta_i p_i$

to those in Algorithm 7. Furthermore the complexity per iteration step increases. Additionally a linear system in every iteration cycle has to be solved. Thus the preconditioner M should have a simple structure to keep the complexity small.

1.4. Generalized Minimum Residual Method

In the following, the goal to solve a non-singular system $Ax = b$, in which A is not necessarily symmetric or positive definite. Thus an extension to the method of Conjugate Gradients is needed which does not work in this case because solving $Ax = b$ is no longer equivalent to minimizing $f(x) = \frac{1}{2}x^T Ax - b^T x$. From the projection methods point of view, it is no longer possible to use the Lanczos algorithm to find a basis of $\mathcal{V}$. Instead, the Arnoldi algorithm has to be employed.

The GMRES approach is to minimize the residual

$$J(w) := ||Aw - b||_2^2. \tag{1.29}$$

Obviously x^* satisfies $J(x^*) = \min J(w) = 0$. Intention is to minimize $J(w)$ with respect to a Krylov subspace $\mathcal{K}_m(r_0, A)$. x_k can be written as $x_k = x_0 + z$ with $z \in \mathcal{K}_k(r_0, A)$. Therefore it yields

$$\begin{aligned} J(x_k) &= \min_{z \in \mathcal{K}_k} J(x_0 + z) \\ &= \min_{z \in \mathcal{K}_k} ||A(x_0 + z) - b||_2^2 \\ &= \min_{z \in \mathcal{K}_k} ||Az + r_0||_2^2. \end{aligned} \tag{1.30}$$

Note that GMRES is a projection method that takes $\mathcal{V} = \mathcal{K}_m(A, r_0)$ and $\mathcal{W} = A\mathcal{K}_m(A, r_0)$ (in notation of Chapter 1.1)[**Fac00**], [**Saa03**]. Thus the Petrov-Galerkin condition must hold,

$$r_k = b - Ax_k \perp A\mathcal{K}_k.$$

In this case the Petrov-Galerkin condition is equivalent to the norm minimizing condition (1.30) [**Fac00**].

Now an orthonormal basis of $\mathcal{K}_k(r_0, A)$ is computed. Therefore the Arnoldi algorithm can be used which is described in Chapter 1.2.1 with the first basis vector v_1 defined as

$$v_1 := \frac{1}{\beta} r_0, \; \beta := ||r_0||_2.$$

Defining $V_k = (v_1|...|v_k)$ with the orthonormal basis vectors v_i, one can write $z = V_k c$ with $c \in \mathbb{R}^k$. With (1.6) the following relations hold

$$\begin{aligned} r_0 - Az &= r_0 - AV_k c \\ &= \beta v_1 - V_{k+1} H_k c \\ &= V_{k+1}(\beta e_1 - H_k c) \end{aligned}$$

and because of the orthogonality of the normalized v_i it can be obtained:

$$J(x_k) = \min_c ||\beta e_1 - H_k c||_2^2. \tag{1.31}$$

c can be computed by solving a least-squares problem. With c it yields $x_k = x_0 + V_k c$. Altogether the algorithm is as shown in Algorithm 10.

Algorithm 10 GMRES algorithm

1: $r_0 = b - Ax_0$, $\beta := ||r||_2$, $v_1 := \frac{1}{\beta} r_0$
2: $\bar{H} := \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$, set $\bar{H} = 0$
3: **for** $j = 1, 2, ..., m$ **do**
4: $\quad w_j := Av_j$
5: $\quad$ **for** $i = 1, .., j$ **do**
6: $\quad\quad h_{ij} := \langle w_j, v_i \rangle$
7: $\quad\quad w_j := w_j - h_{ij} v_i$
8: $\quad h_{j+1,j} = ||w_j||_2$. If $h_{j+1,j} = 0$ set $m := j$ and go to 10
9: $\quad v_{j+1} = \frac{1}{h_{j+1,j}} w_j$
10: $x_m = x_0 + V_m y_m$ with y_m minimizes $||\beta e_1 - \bar{H}_m y||_2$

Now the least-squares problem (1.31) is addressed. Therefore $H_k$is transformed into an upper triangular form using Givens-Rotations. Recall, a Givens-Rotation is performed by multiplying a rotation matrix of the form

$$\Omega_{ij} = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & & s_i & & \leftarrow i \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & & & & 1 & \\ & & & -s_i & & c_i & & \leftarrow j \\ & & & & & & & 1 \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}$$

with $c_i^2 + s_i^2 = 1$. Here the rotation matrices $\Omega_j := \Omega_{j,j+1}$ is used to transform the entries $h_{j+1,j}$ of H to zero. Ω_j can be computed by

$$s_j = \frac{h_{j+1,j}}{\sqrt{h_{j+1,j}^2 + h_{j,j}^2}}, \qquad c_j = \frac{h_{j,j}}{\sqrt{h_{j+1,j}^2 + h_{j,j}^2}}. \tag{1.32}$$

By applying the rotation matrices to the least-squares problem it ensues

$$\hat{R}_k = \begin{pmatrix} R_k \\ 0 \end{pmatrix} \in \mathbb{R}^{k+1 \times k}, \qquad \hat{d}_k = \begin{pmatrix} d_k \\ \gamma_k \end{pmatrix} \in \mathbb{R}^{k+1}, \gamma_k \in \mathbb{R},$$

where R_k is an upper triangular matrix and

$$\hat{R}_k = \Omega_k \Omega_{k-1} \cdots \Omega_1 H_k = Q_k H_k,$$
$$\hat{d}_k = \Omega_k \Omega_{k-1} \cdots \Omega_1 \beta e_1 = Q_k \beta e_1.$$

Note that this transformation works for every matrix $H \in \mathbb{R}^{n \times N}$ with rank $N < n$, see [**SK06**]. Solving the least-squares problem is equivalent to solve the linear system $R_k c = d_k$. Using the orthogonality of Q_k respectively of the Ω_i it holds

$$J(x_k) = ||r_k||_2^2 = \min ||\beta e_1 - H_k c||_2^2 = \min ||\hat{d}_k - \hat{R}_k c||_2^2 = \gamma_k^2.$$

Thus the residual norm can be determined without computing r_k and x_k or solving the least-squares problem in each iteration step, [**Saa03**][**SK06**].

For building H_{k+1}, a new column with $k+1$ values is added to the end of H_k by running the Arnoldi algorithm. For this purpose, the rotation matrices $\Omega_1, ..., \Omega_k$ can be used again for transforming H_{k+1}. More precise $\Omega_1, ..., \Omega_k$ have to be applied to the new column of H_{k+1}. For eliminating the entry $h_{k+1,k}$, a new rotation matrix Ω_{k+1} is needed. Applying Ω_{k+1} to the extended right hand side provides

$$\gamma_{k+1} = -s_k \gamma_k, \tag{1.33}$$

for more details see [**Saa03**][**SK06**]. Now it can be seen that the squared residual norm decreases monotonously.

Algorithm 10 is not feasible for large j, because all basis vectors $v_1, ...v_{j-1}$ have to be stored as well as to compute and orthogonalize v_j against them. In addition, the complexity increases linear. Furthermore from section 1.2.1 it is known that rounding errors in the Arnoldi algorithm destroy the orthogonality of the $v_1, ..., v_j$ for large j. As a solution to these problems, one can run $m << n$ iteration cycles, compute x_m and afterwards restart the algorithm with initial guess x_m. This is called restarted GMRES or GMRES(m) as shown in Algorithm 11. m indicates the number of iteration cycles respectively the dimension of the Krylov subspace.

Algorithm 11 GMRES(m) algorithm

guess x_0
for $l = 1, 2, ...$ **do**
  $r_0 = b - Ax_0$, $\beta := ||r||_2$, $v_1 := \frac{1}{\beta} r_0$
  while $j \leq m$ and $||res|| > \epsilon$ **do**
    $w_j := Av_j$
    for $i = 1, .., j$ **do**
      $h_{ij} := \langle w_j, v_i \rangle$
      $w_j := w_j - h_{ij} v_i$
    $h_{j+1,j} = ||w_j||_2$
    $v_{j+1} = \frac{1}{h_{j+1,j}} w_j$
    for $i = 1, .., j-1$ **do**
      apply old rotation Ω_i to the last column of H_j
    compute and apply new rotation Ω_j
    compute new residual res
  solve $R_l c_l = d_l$, compute $x_l = x_0 + V_l c_l$, $x_0 = x_l$
  stop if $||res|| < \epsilon$

There are different truncated GMRES versions. For example the Arnoldi algorithm can be replaced by a incomplete orthogonalization process. More information about such versions can be found in [**Saa03**].

The complexity of restarted GMRES is discussed due to implementation issues. First note, that the complexity of GMRES increases from iteration cycle to iteration cycle. This is caused by the basis orthogonalization within the Arnoldi algorithm, where one (sparse) matrix-vector multiplication, $k+1$ vector updates and $k+1$ dot products have to be performed in the kth iteration step. In GMRES(m) the parameter k is at most m with $m << n$. Therefore a complexity of $O(nnz)$ can be obtained in every iteration cycle for GMRES(m).

In the previous sections the question what could happen if $h_{j+1,j} = 0$ within the Arnoldi-Algorithm in a certain step j was not answered. This is the only possible reason for a breakdown of GMRES. From $h_{j+1,j} = 0$ follows by (1.32) and (1.33) that $||r_j||_2 = 0$ and the exact solution x^* is found. Furthermore it is proved, that if the algorithm stops with the exact solution x^*, then $h_{j+1,j} = 0$, see [**Saa03, SK06**].

Since there are at most n independent vectors v_j in $\mathbb{R}^n$, the general GMRES algorithm 10 computes the solution x^* after at most n steps. However as mentioned above, this is only a theoretical result. In practice it is often not possible to run n iterations, even if rounding errors in the Arnoldi algorithm are ignored. Thus the focus is set on GMRES(m) in the following, aware of the fact that there is (as far as the author knows) no proof for general convergence of GMRES(m), see [**SK06**]. However with several assumptions to the matrix A there are convergence results. For example, for a positive definite matrix A GMRES(m) converges for any $m \geq 1$. A proof can be found in [**Saa03**].

Important can be an upper bound for the convergence rate as for the CG- Method in 1.28. In the following only results are presented without corresponding proofs which can be found in [**Saa03**]. The matrix A is henceforth assumed to be diagonalizable, that is $A = XDX^{-1}$, with all eigenvalues located in an ellipse E. Then it yields

$$||r_m||_2 \leq \kappa_2(X) \frac{T_m(a/d)}{|T_m(c/d)|} ||r_0||_2$$

where r_m is the residual in the m-th GMRES step and T_m denotes the Teschebyscheff polynomial of degree m. The constant c describes the center, d the focal distance and a the major semi axis of the ellipse E.

For preconditioning the GMRES algorithm all three strategies presented in section 1.5 can be used. Focus in this work is on right preconditioning to avoid modifications of the residual. Algorithms with left and symmetric preconditioning can be found in [**Saa03**].

Intention in to solve

$$AM^{-1}u = b, \qquad x = M^{-1}u$$

with some kind of preconditioner M. Therefore the Arnoldi algorithm computes an orthonormal basis of the Krylov subspace

$$\text{span}\{r_0, AM^{-1}r_0, ..., (AM^{-1})^{m-1}r_0\}.$$

The GMRES algorithm computes the solution $u_m = u_0 + V_m c_m$ of $AM^{-1}u = b$. The original solution x_m is computed by multiplying M^{-1}

$$x_m = M^{-1}u_m = x_0 + M^{-1}V_m c_m.$$

Hence one obtains the right GMRES algorithm 12.

Finally a look is taken at a special preconditioning for GMRES. This technique allows to change the preconditioner in every step and is therefore called flexible GMRES or FGMRES. Line 5 in Algorithm 12 is split into two steps, first apply the preconditioner M_j^{-1} to the last basis vector v_j and afterwards apply A to the

Algorithm 12 GMRES(m) algorithm with right preconditioning

guess x_0
for $l = 1, 2, ...$ **do**
  $r_0 = b - Ax_0$, $\beta := ||r||_2$, $v_1 := \frac{1}{\beta} r_0$
  for $j = 1, 2, ..., m$ **do**
    $w_j := AM^{-1}v_j$
    for $i = 1, .., j$ **do**
      $h_{ij} := \langle w_j, v_i \rangle$
      $w_j := w_j - h_{ij}v_i$
    $h_{j+1,j} = ||w_j||_2$
    $v_{j+1} = \frac{1}{h_{j+1,j}} w_j$
  solve $\min ||\beta e_1 - H_l c_l||_2^2$, compute $x_l = x_0 + M^{-1}V_l c_l$, $x_0 = x_l$
  Stop if $||r||_2$ small enough

preconditioned vector z_j. For computing the approximate solution x_m the preconditioned vectors z_j have to be saved. The relation (1.4) changes for FGMRES

Algorithm 13 FGMRES(m) algorithm

guess x_0
for $l = 1, 2, ...$ **do**
  $r_0 = b - Ax_0$, $\beta := ||r||_2$, $v_1 := \frac{1}{\beta} r_0$
  for $j = 1, 2, ..., m$ **do**
    $z_j := M_j^{-1}v_j$
    $w_j := Az_j$
    for $i = 1, .., j$ **do**
      $h_{ij} := \langle w_j, v_i \rangle$
      $w_j := w_j - h_{ij}v_i$
    $h_{j+1,j} = ||w_j||_2$
    $v_{j+1} = \frac{1}{h_{j+1,j}} w_j$
  solve $\min ||\beta e_1 - H_l c_l||_2^2$, compute $x_l = x_0 + Z_l c_l$, $x_0 = x_l$
  Stop if $||r||_2$ small enough

to

$$AZ_m = V_{m+1}H_m.$$

Note that the convergence of FGMRES cannot be proved. For more information see e.g. [**Saa03**].

1.5. Preconditioning of Linear Systems

The convergence of iterative solvers depends usually on the condition number $\kappa(A)$ of the matrix. Improvements of the convergence speed can be achieved by transforming the original system $Ax = b$ to an equivalent problem $\tilde{A}\tilde{x} = \tilde{b}$ with $\kappa(\tilde{A}) < \kappa(A)$. This approach is called preconditioning and there are different transformations possible like left (1.34), right (1.35) or symmetric (1.36) preconditioning.

A left preconditioning can be performed by choosing an adequate and non-singular matrix M and multiply $Ax = b$ by M^{-1}

$$M^{-1}Ax = M^{-1}b. \tag{1.34}$$

Solving the transformed linear system can be done explicitly by computing $M^{-1}A$, $M^{-1}b$ and using an iterative solver or implicit by using the preconditioner M within

an algorithm. Implicit preconditioning is used in the PCG-Algorithm (see Chapter 1.3) and in preconditioned GMRES (see Chapter 1.4). Remark, that the left preconditioning changes the residual r_j of original system to

$$\tilde{r_j} = M^{-1}b - M^{-1}Ax_j = M^{-1}(b - Ax_j) = M^{-1}r_j.$$

With a right preconditioning it is possible to avoid a modification of the original residual. For this purpose the solution x is modified.

$$AM^{-1}y = b, \qquad \text{with } y = Mx. \tag{1.35}$$

The residual satisfies

$$\tilde{r_j} = b - AM^{-1}y_j = b - AM^{-1}Mx_j = b - Ax_j = r_j.$$

Hence one can work with the original residual in the preconditioned system. An optimal choice for M is A, since

$$\kappa(AA^{-1}) = \kappa(A^{-1}A) = \kappa(I) = 1.$$

As a result, the non-singular matrix M^{-1} should approximate A^{-1} in some sense. Both preconditioning methods are not suitable for the method of conjugate gradients, because left and right preconditioning do not preserve the structure of A. Therefore a method is needed, which preserves symmetry and the positive definite structure that are essential for the CG solver. The solution is a combination of right and left preconditioning. Define $C^{-T} := (C^{-1})^T$ and let A be positive definite and symmetric. The original System is transformed by setting

$$\begin{aligned} \tilde{A} &:= C^{-1}AC^{-T} \\ \tilde{x} &:= C^T x \\ \tilde{b} &:= C^{-1}b. \end{aligned} \tag{1.36}$$

The resulting matrix $\tilde{A}$ is also positive definite and symmetric, because

$$\begin{aligned} \tilde{A}^T &= (C^{-1}AC^{-T})^T = C^{-1}A^TC^{-T} = \tilde{A} \\ x^T\tilde{A}x &= x^TC^{-1}AC^{-T}x = (C^{-T}x)^TA(C^{-T}x) > 0, x \neq 0. \end{aligned}$$

Furthermore $\tilde{A}$ is similar to the matrix $M^{-1}A$ with $M := CC^T$, since

$$C^{-T}\tilde{A}C^T = C^{-T}C^{-1}A = (CC^T)^{-1}A = M^{-1}A.$$

From linear algebra it is known, that the eigenvalues of $\tilde{A}$ are similar to the eigenvalues of $M^{-1}A$. For the condition number of $\tilde{A}$ it holds

$$\kappa(\tilde{A}) = \frac{\lambda_{\max}(M^{-1}A)}{\lambda_{\min}(M^{-1}A)}.$$

Likewise to left and right preconditioning the best choice for M is A. But therefore a Cholesky decomposition of A has to be calculated, which is corresponding to solve the linear system. One should choose M^{-1} to approximate A^{-1} in order to take advantage of preconditioning.

Below two common preconditioners are considered in detail. First Jacobi preconditioning and afterwards the incomplete LU-decomposition is considered.

1.5.1. Jacobi and Block-Jacobi Preconditioning. The Jacobi approach is to set $M := \text{diag}(A)$. Applying M^{-1} to a vector x corresponds to multiply every

entry x_i of x with $\frac{1}{a_{ii}}$, where a_{ii} is the i-th diagonal element of A. Hence the diagonal of A is scaled to one.

$$M := \begin{pmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{pmatrix}$$

Only little storage is needed to save M. It is even possible to use no extra storage by computing the entries "on the fly", which means that they are computed when they are needed. Using the Jacobi preconditioner in parallel solvers is easy, because applying the preconditioner to a vector needs no communication. However the benefit of this preconditioning depends strongly on the underlying application. Based on practical observations, a benefit from Jacobi preconditioning can be expected when the matrix has a small bandwidth or is diagonal dominant.

For a symmetric and positive define matrix with constant diagonal entries, the Jacobi preconditioning scales the eigenvalues of A with a constant factor. Thus the condition number of A does not change. As a result Jacobi preconditioning is without effect on the convergence and only causes additional computation costs.

By assuming a linear system $Ax = b$, where A is symmetric, positive define and $a_{i,i} = n \in \mathbb{R}$. Therefore the preconditioner M is

$$M = \begin{pmatrix} n & & \\ & \ddots & \\ & & n \end{pmatrix}, \qquad M^{-1} = \begin{pmatrix} \frac{1}{n} & & \\ & \ddots & \\ & & \frac{1}{n} \end{pmatrix}.$$

Note that $\lambda(M)_i = n$ and $\lambda(M^{-1})_i = \frac{1}{n}$. The condition number of A is $\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$, where $\lambda_{max}(A)$ ($\lambda_{min}(A)$) denotes the largest (smallest) eigenvalue of A. Now the effect of Jacobi preconditioning on the condition number is analyzed. It yields

$$M^{-1}Ax = M^{-1}\lambda x = \frac{\lambda}{n}x$$

for all eigenvalues λ and corresponding eigenvectors x of A and therefore $\kappa(A) = \kappa(M^{-1}A)$. In the CG-algorithm it can be seen that the factor $\frac{1}{n}$ is canceled and the Jacobi-preconditioned version has the same solution approximations x_k and residuals r_k as the un-preconditioned CG.

One example for such a case is the 2D Poisson equation on a unit-square that is solved by using and finite difference discretization based on a 5-point-stencil. Numerical results showing the effect of useless Jacobi preconditioning can be found in Figure 4.15 in Chapter 4.9.2.

An expansion of the Jacobi preconditioner is the so called Block-Jacobi preconditioner, where not only the diagonal elements are taken into account but bigger blocks (Figure 1.2). On the blocks B_i any solver can be applied, e.g. a ILU (see Chapter 1.5.2).

Due to the fact that the blocks of the Block-Jacobi are independent to each other, in terms of degrees of freedom, the Block-Jacobi takes not all couplings of the problem into account when more then one block is used. Therefore the parallelization of Block-Jacobi preconditioners is trivial and no communication is needed. In dependence of the block size and the performed solver on the blocks, the number of iterations can be reduced in comparison to the pure Jacobi (see Chapter 4.9.2).

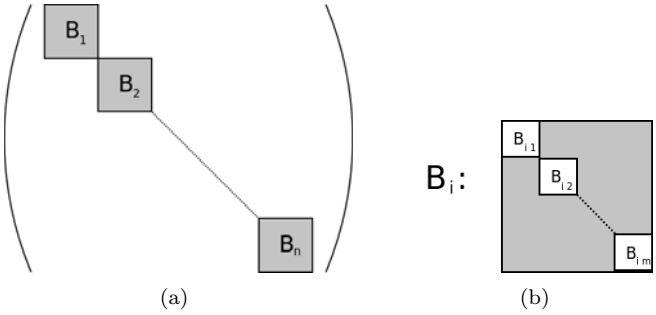


Figure 1.2. a) Blocks of a Block-Jacobi-preconditioner applied to a matrix. b) Nested Block-Jacobi preconditioner.

1.5.2. Incomplete LU-Factorization. The main idea of ILU is to approximate the matrix A by $A = LU - R$, where R is an error matrix, L a lower triangular matrix and U an upper triangular matrix. Therefore some fill in elements are dropped during the factorization process of A. Compared to the Jacobi preconditioning assembling and applying the ILU preconditioner M defined as $M := LU$ is much more involved. Applying M^{-1} to a vector x means solving the linear system $LUy = x$ by forward and backward substitution.

A general approach is to give an index set P in advance, which describes the nonzero structure of L and U. The decomposition is done by Gaussian elimination and only entries of L and U with indices in P are saved. Other entries are set to zero. Therefore L and U are not necessarily regular. But the existence is guaranteed if A is a M-matrix. For detailed theory the author refers to [**Saa03**].

Choosing P_A as the nonzero indexes in A leads to the factorization called ILU(0). This means, that no fill-in in L and U is allowed. The nonzero structure of L and U is equal to the lower respectively the upper triangular part of A. Hence ILU(0) is easy to use with the CSR matrix format (see Chapter 4.1), because the *col* array and *row* array of $[LU]$ are equal to the arrays of A and only an additional *val* array is needed to store the non-zeros of $[LU]$. In practice L and U are stored in a single CSR- matrix $[LU]$. Since the diagonal values of L equal one, they are not saved explicitly. This approach is also used in the following ILU algorithms.

$$[LU] = \begin{pmatrix} & U \\ L & \end{pmatrix}$$

Figure 1.3. L and U stored in a single matrix

Algorithm 14 ILU(0)

1: **for** $i = 1, ..., n-1$ **do**
2: **for** $k = 0, ...i-1$ and $(i,k) \in P_A$ **do**
3: $a_{ik} := a_{ik}/a_{kk}$
4: **for** $j = k+1, ..., n-1$ and $(i,j) \in P_A$ **do**
5: $a_{ij} := a_{ij} - a_{ik}a_{kj}$

An obvious approach is to allow fill-in during the factorization. This leads to decompositions called ILU(p), where p denotes the *level of fill*. Before defining the

level of fill the idea of computing ILU(p) is shown. The nonzero structure of ILU(1) is defined by the nonzero structure of the product L_0U_0, where L_0 and U_0 are taken from ILU(0). The nonzero structure of ILU(2) is defined analogue by taking the structure of ILU(1), and so on. For computing ILU(p) one has to perform an ILU(0) factorization of A with the nonzero structure of ILU($p-1$), whereas at first the fill in positions in A are set to zero. Fortunately, it is not required to compute the previous nonzero structures for building ILU(p). Hence the level of fill is needed. Every matrix entry a_{ij} is related to a level of fill lev_{ij} [**Saa03**]. Initially, it is defined as

$$lev_{ij} := \begin{cases} 0, & \text{if } a_{ij} \neq 0 \text{ or } i = j \\ \infty, & \text{otherwise.} \end{cases} \tag{1.37}$$

During the Gauss elimination the level of fill for every modified matrix entry is updated by [**Saa03**]

$$lev_{ij} = \min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}. \tag{1.38}$$

This strategy induces the following dropping rule. Every matrix entry with level of fill greater than p is deleted (set to zero) during the factorization. In an implementation ∞ can be replaced by $p+1$, due to the fact, that the level of fill will never decrease. A main problem of ILU(p) is the memory allocation in advance. It is not possible to predict a priori the number of nonzero entries in ILU(p). ILU(p)

Algorithm 15 ILU(p)

1: define the level of fill for all matrix elements using (1.37)
2: **for** $i = 1, ..., n-1$ **do**
3: **for all** nonzero elements before the diagonal element in row i of A **do**
4: $a_{ik} := a_{ik}/a_{kk}$
5: update level of fill of a_{ik}
6: **for all** nonzero elements after the diagonal element in row i of A **do**
7: $a_{i\star} := a_{i\star} - a_{ik}a_{k\star}$
8: update level of fill of $a_{i\star}$
9: replace all elements in row i with $lev(a_{ij}) > p$ by zero

is a structural approximation to the LU- decomposition of a matrix A. It does not take the values been dropped into account. A small value may be kept, while a large value is dropped. It would be more desirable to drop small values and keep large values. This leads to strategies using "drop tolerances" like ILUT (ILU with threshold) [**Saa03**]. [**Saa03**] proposes two dropping rules. First elements with absolute value smaller than a tolerance τ are dropped and to control memory usage, only the p largest values of a row are kept. The ILUT factorization exists for diagonal dominant matrices under certain conditions. For detailed theory of ILUT the author refers to [**Saa03**]. Note that algorithms 14, 15 and 16 do not use pivoting and hence they could break down. Now consider some implementation issues. Algorithms 15 and 16 use a full row array of length n to handle fill in. This array is initialized with zeros. Afterwards the nonzero elements of row i in A are copied into this array. Then the Gaussian elimination adds the previous sparse rows to the full row by eliminating entries. Before starting the elimination for the next row the full row is transferred to CSR- structure and added into the matrix [LU].

In [**MAK03**] is proposed a different approach to setup a ILU preconditioner. In this approach a set P of nonzero indexes is computed in advance by a Boolean matrix strategy. Based on this set, it can be competed a ILU preconditioner with a algorithm similar to Algorithm 14.

Algorithm 16 ILUT(τ,p)

```
1: for i = 1, ..., n − 1 do
2:   for all nonzero elements before the diagonal element in row i of A do
3:     a_ik := a_ik / a_kk
4:     drop (set to zero) a_ik if |a_ik| < τ · ||a_i*||_2
5:     if a_ik ≠ 0 then
6:       for all nonzero element after the diagonal element in row i of A do
7:         a_i* := a_i* − a_ik a_k*
8:   drop (set to zero) a_i⋆ if |a_i⋆| < τ · ||a_i*||_2
9:   keep only the p largest elements in L and the p largest elements in U
```

Implementation issues regarding the parallelization of the forward and backward substitution can e.g. be found in [**GK95**].

1.6. Stopping Criteria for Iterative Solvers

In the previous sections it is not described how to stop a iterative solvers, in case of convergence or divergence. In this section a brief survey of some criteria is given. More details can be found in [**CP09**],[**ADR92**] or [**BBC$^+$94**].

Ideally a stopping criterion should identify if the error $e_j = x_j - x^*$ is small enough, check for divergence and also stop if the error is no longer decreasing. In general it is not possible to determine the error in absence of the correct solution, hence the residual is often used instead. Some simple stopping criteria for a given tolerance *tol* are

$$||e_k|| \leq tol \cdot ||e_0|| \qquad \textit{relative error norm criterion},$$
$$||r_k|| \leq tol \cdot ||r_0|| \qquad \textit{relative residual norm criterion},$$
$$||r_k|| \leq tol \qquad \textit{absolute residual norm criterion}.$$

The residual norm criterion depends strongly on the initial guess x_0. Additionally one can use an integer *maxit* to give an upper bound for the number of iterations. Note that the tolerance *tol* should be greater than the machine precision ϵ. The following relation between the error norm and the residual norm holds

$$||e_k|| \leq ||A^{-1}|| \cdot ||r_k||.$$

Note that the error norm need not to be smaller than residual norm, but if one knows something about $||A^{-1}||$ it is possible to estimate the error.

There are also several stopping criteria based on backward error analysis. The norm wise backward error is defined as $\max(\frac{||\delta A||}{||A||}, \frac{||\delta b||}{||b||})$ related to the exact solution of $(A+\delta A)\hat{x} = b+\delta b$. The terms δA and δb describe perturbations to the linear system $Ax = b$. In this context the following stopping criteria proposed in [**ADR92**] and [**BBC$^+$94**] are:

$$||r_k|| \leq tol \cdot (||A|| \cdot ||x_k|| + ||b||),$$
$$||r_k|| \leq tol \cdot ||b||.$$

For more details on stopping criteria and their theory the author refers to the literature.

CHAPTER 2

New Technologies in Computer Architecture

A short overview of types of parallelism as well as abstract hardware representations of parallel architectures and detailed information regarding modern hardware technologies like GPUs and FPGAs can be found in the first section. Following behind, the most common programming paradigms MPI, OpenMP, OpenCL, CUDA and PGAS languages as well as the hardware description language (VHLD) are exposed. Parts of this chapter are similar to [**Hah09**] and [**Gal10**].

2.1. Parallel Architectures

2.1.1. Types of Parallelism. *Bit-level parallelism* is the most elementary type of parallelism, meaning the amount of Bits is processed by a processor in parallel. Current multiprocessors (see 2.1.3) normally have a computer *word size* of 64 bits while previous generations of processors have often used 32 bit words. A second type of parallelism is the *instruction level parallelism*, formally known as *pipelining*. For this purpose different instructions in a stream are reordered, so that several instructions in a pipeline (stream) overlap. Parallelizing loops leads often to *data parallelism*, which means that several processing units work in parallel with the same instruction on different data. Finally there is a *process* or *thread parallelism* where different tasks are processed in parallel. For detailed information see e.g. [**HP07**] and [**RR07**].

2.1.2. Classification of Computer Architectures. The presented model for categorizing computer architectures was proposed by Michael J. Flynn in 1966 [**Fly72**]. His approach is to group all computers into four categories concerning their instruction and data parallelism.

	Single Data	**Multiple Data**
Single Instruction	SISD	SIMD
Multiple Instructions	MISD	MIMD

Figure 2.1. Flynn's taxonomy. Computers are classified with respect to instruction and data parallelism.

In the following, all four combinations of data and instruction parallelism are explained:

SISD: A common uniprocessor, a processor with only one core, is an example for this category. This processor can apply only one instruction to one single data stream at a certain time.

SIMD: The typical example for SIMD architectures are vector processors. A set of processors using different data streams executes the same instruction. The data streams are processed in parallel (*data-level parallelism*).

MISD: This is only a theoretical category. There exist no computers with this design.

MIMD: There are several processors with their own instruction streams and their own data. Common multi-core processors are of this type, where several threads work in parallel (*thread-level parallelism*). High performance computing clusters are often of this type, as well. Furthermore MIMD architectures can be divided into two classes concerning the memory organization. This classes are shared memory architectures and distributed memory architectures, which are defined via memory addressing.

More information on Flynn's classification can be found in [**Fly72**] and [**HP07**].

2.1.3. Multiprocessor Systems. Microprocessors usually aim for fast treatments of computations of a single thread. Therefore multiple techniques are employed, like speculative execution, pipelining and others. This leads often to chips where the arithmetic units allocate only a minor area of the core die area. About one decade ago (view from 2010) limitations in chip design, like leakage current and power dissipation led to processors made up of multiple conventional microprocessors, which are called multi-core or many-core processors.

Multiprocessor systems use more than two central processing units (CPU) within a single system. A CPU, which consists of at least two processor cores, is called a multi-core processor. More details on multiprocessors and multi-core processors as well as their specific design can be found in [**HP09, BBC$^+$08**]. Nowadays (2010) multi-core processors can be found in almost every desktop computer as dual-, triple-, quad- or hexa-core processors. The general trend in processor development goes to processors with dozens or even hundreds of cores. Usually, processors can differ significantly in their functionality, for example in the size of the instruction set or in the levels and sizes of their caches. Below we want to distinguish multiprocessor systems with respect to their memory organization.

A *shared memory* architecture is defined as a group of processors sharing a single memory in the sense that all processors are working within a single memory address space, whereas it is possible that the memory is physically distributed. It is important that every processor can access every memory location. Thus different processors can communicate via shared variables in memory.

A *thread* can be defined as a runtime entity that is able to independently execute a stream of instructions in a shared memory environment [**CJvdP08**]. Different threads share data and resources, but may also have some private data. Multiprocessors with a single physical memory and a single memory address space are called *symmetric shared memory multiprocessors* (SMPs), see Figure 2.2 with shared memory address space. The related architecture is also called *uniform memory access* (UMA) architecture, because all processors have a uniform latency for memory accesses.

Multiprocessors with a physically distributed memory and a shared memory address space are called *distributed shared memory* (DSM) architectures. Usually this architecture offers a non-uniform memory access (NUMA), because the access time depends to the data location in memory. For both cases, symmetric and distributed shared memory, the problem of *cache coherence* arises. This is when different processors respectively threads load the same data from main memory in their local caches, modify the data in different ways and write them back to the main memory. Therefore strategies are needed to avoid such situations. Note that caches are no shared memory. Detailed theory on shared memory architectures and on the cache coherency can be found in [**HP07**] or [**HP09**].

A *distributed memory* architecture offers a virtually (and usually physically) distributed memory address space. In contrast to the shared memory systems the address spaces for each process are disjoint meaning that the global address space is divided into several private address spaces. Hence a process respectively processor cannot access the memory of another process directly. For this purpose we need explicit message passing between the related address spaces respectively the processors. Hence we need dedicated routines on a process to send and receive data. Figure 2.3 with distributed address space shows the structure of a distributed memory multiprocessor. Figure 2.2 can also be seen as a architecture with a distributed memory address space, by assuming the main memory divided into different address spaces. In a distributed memory architecture multiple independent instruction streams are called *processes*. In contrast to threads, different processes are independent, so they share no resources and work in private address spaces. Threads exist as a subset of a process.

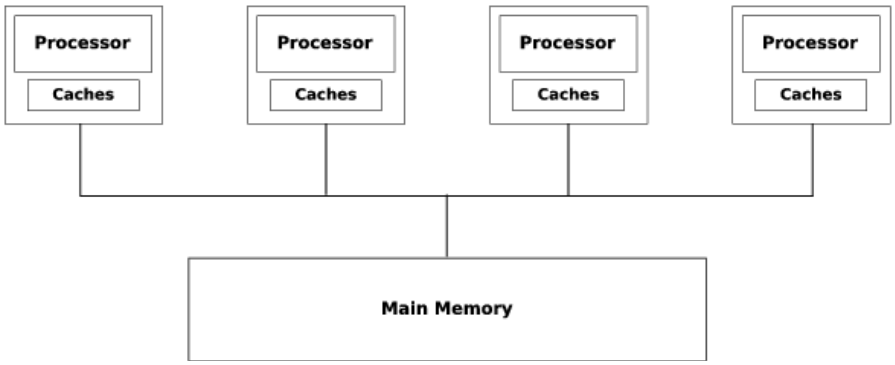


Figure 2.2. Structure of a multiprocessor with physically shared memory. The memory address space can be divided or shared.

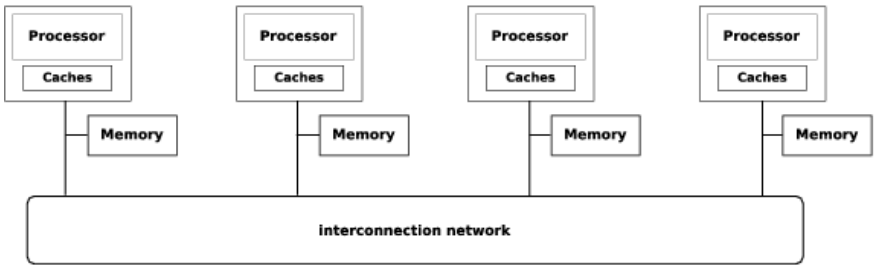


Figure 2.3. Structure of a multiprocessor architecture with physically distributed memory. The memory address space need not to be divided, it can also be shared.

2.1.4. Graphic Processing Units. Graphic Processing Units are special multi-core systems (often called many-core systems), whereas the cores have only a very limited instruction set compared to most CPUs. Originally GPUs were developed for graphic applications like multimedia and the computer games. In the past GPUs had only single precision processing units since the accuracy was sufficient for visualizing data. Rounding errors or other effects influencing the quality of an image that is often only $\frac{1}{25}$ second on the screen did not play an important role. With the introduction common CPU properties like full floating-point support, IEEE 754 rounding and built-in double precision support GPU technology became more interesting for the High Performance Computing (HPC) market. That there exist

multiple methods to overcome the absence of previously mentioned properties (by assuming that the GPU is connected to a host offering such properties) is shown in Chapter 3.

Modern GPUs consist of many parallel processors and are capable to run many concurrent threads. They based on a stream processing design, which is a series of operations are applied to a data stream. Figure 2.4 shows the hardware setup of a NVIDIA GT200 respectively T10 GPU. It consists of 10 *Thread Processing Clusters* (TPC), in which each TPC is made up of three *streaming multiprocessors* (SM) and each SM contains eight processor cores. This cores are called *streaming processors* (SP) or thread processors. Altogether there are 240 (single precision) processor cores and 30 double precision units on a single chip. The execution across the TPCs works in MIMD style, while execution across each SM works in SIMD style. NVIDIA describes the execution across SMs as *single instruction multiple thread* (SIMT). More information on the NVIDIA GT200 GPU can be found in [**NVI08**]. Meanwhile the successive generation of Nvidias GPUs in form of the GF100 respectively T20 chip is available (see Chapter 5.2 for more information and a performance comparison). Main structural characteristics stayed the same but it can be assumed that this will not be the case for future GPUs. *General purpose computing on graphics processing units* (GPGPU) means running tasks on a GPU, which normally are handled by a CPU. The main manufacturers of high performance graphics chips, NVIDIA and AMD, offer specific programming languages to process data streams beside off graphics rendering on their products. In the past AMD distributed *ATI Stream* including *Brook+* but went to OpenCL a short time ago. NVIDIA offers with CUDA an extension to the C programming language. A more general approach to take advantage of coprocessor technologies is *OpenCL*. Both OpenCL and CUDA are explained in more detail in Chapter 2.2.3.

For more information on graphics processing units and their capabilities see e.g. [**AHHRed**], [**Hah09**] or [**HP09**].

2.1.5. Reconfigurable Architectures. In the recent years more and more heterogeneous and hybrid hardware platforms appear on the market for high performance computers. Beside the GPU-accelerated platforms, which have been considered in the previous section, systems employing FPGAs (Field-Programmable Gate Array) are offered by multiple vendors.

FPGAs are integrated circuits offering the possibility to be configured by the user after the manufacturing process. The hardware can be configured for a specific problem and enables programmers to make as many operations run concurrently as they can physically fit on the FPGA. By doing so it can be possible to use the available hardware resources in a very efficient way and, depending on the application, conventional CPUs can be outperformed (see [**SG06**]). This is often the case when computations based on integers have to be performed like in the most common fields of applications for FPGAs, which are digital signal processing, cryptography, bio-informatics and hardware emulation. A configuration of a FPGA is not fixed for all times but it can be reconfigured over and over again. Important is, that this process takes time before the computation can start.

Usually FPGAs are programmed by using a so called hardware description language (HDL). Verilog and VHDL are the most widespread representatives but the usage is more complicated compared to traditional high-level programming languages like Fortran, Java or C/C++. There are converters available to create HDL-code from high-level programming languages, see [**Sch10**] for details. Further information related to FPGAs can be found in [**GG05**].

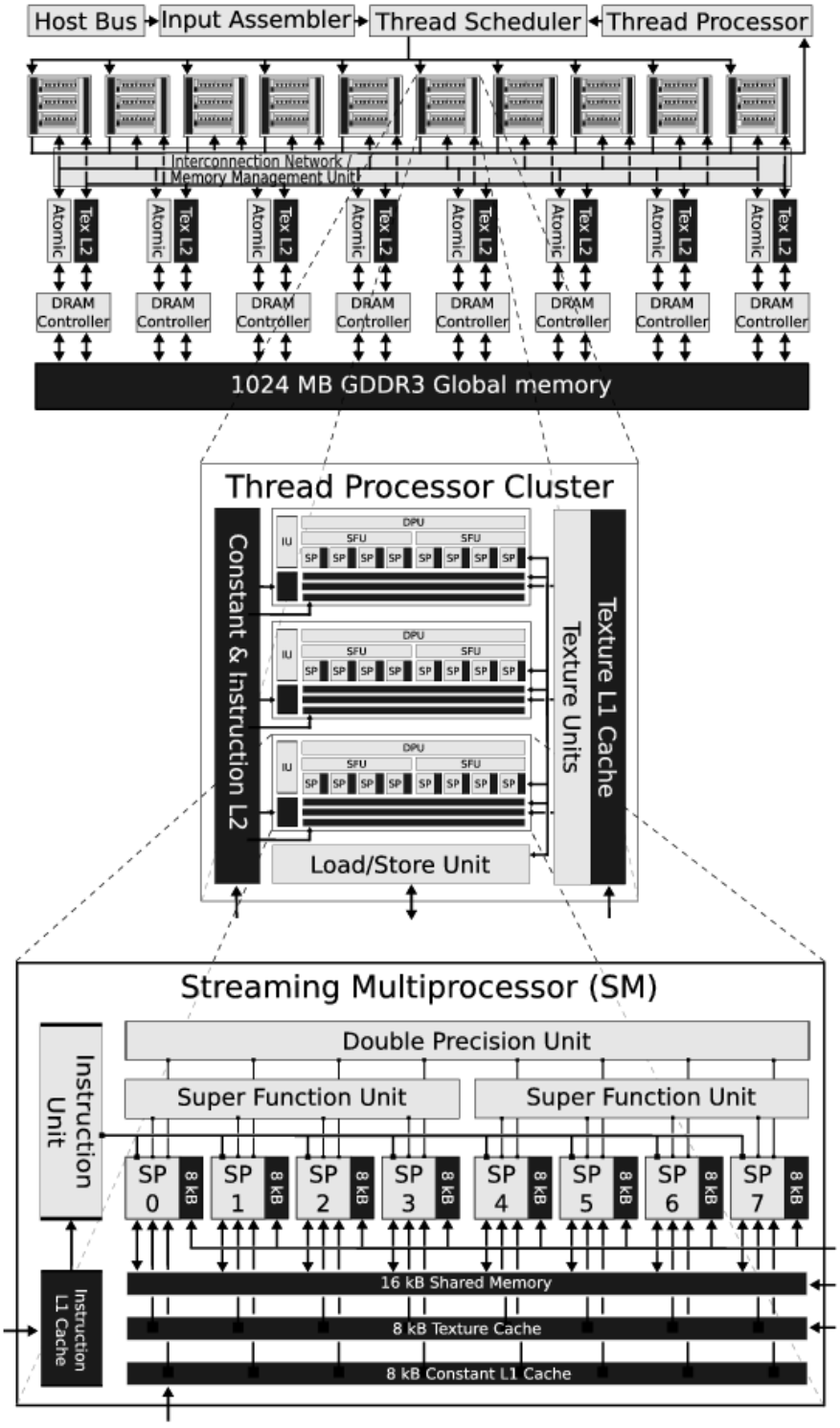


Figure 2.4. Architecture of the NVIDIA GT200 respectively T10 GPU. Picture taken from [**Hah09**].

There are numerous variants of hardware architectures consisting of FPGAs. One representative of a hybrid CPU-FPGA machine is the Convey HC-1 which is a combination of a traditional two-socket computing node (in the following called

host) and a FPGA-equipped accelerator part (in the following called device). A schematic representation of the machine is shown in Figure 2.5. One socket of the host is equipped with a x86-based Intel Xeon processor and the second socket hosts an interconnect to the device. Four Xilinx Virtex5 FPGAs as well as eight memory controllers and sixteen DDR2 slots can be found on the device site. The address space for of the full machine is global, which means that processes running on the host-site can access data in the physical memory of the device part and the other way round. An important remark hereby is that the different access times lead to significantly different performance results, as Chapter 2.1.5 shows. On the software site Convey offers the possibility to use self-written implementations in Verilog as well as existing kernels applicable to be used on the device with a interface syntax that is similar to other libraries (e.g. BLAS etc.). Very simple loops, written in C and without data dependencies, can be compiled for the device automatically by tagging the relevant code using special commands, named pragmas, which are alike to those used for OpenMP.

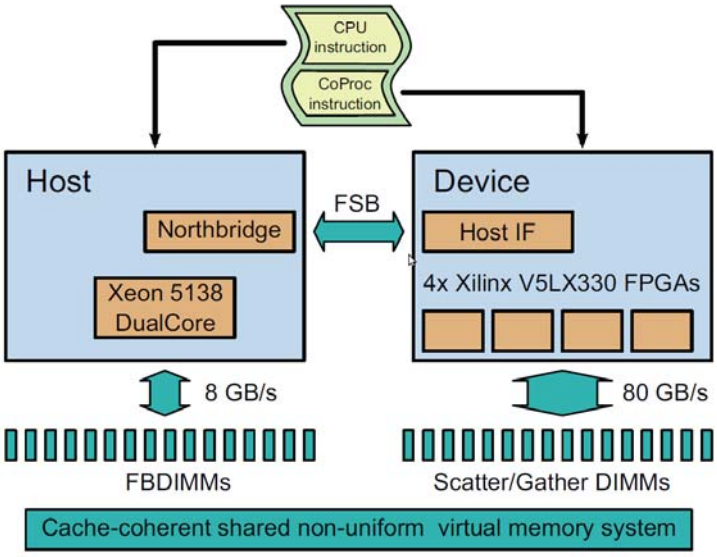


Figure 2.5. Architecture of the Convey HC-1 taken from [**AHW10a**].

A performance evaluation of elementary kernels and a linear solver on the above described hybrid CPU-FPGA architecture is presented in Chapter 5.3 and related work can be found in [**Bre10**] and [**AHW10a**].

2.2. Parallel Programming Paradigms

2.2.1. OpenMP. One of the most common shared memory programming models is OpenMP (Open Multi-Processing). More precisely, OpenMP is a shared memory application programming interface (API) that can be used with the programming languages Fortran, C and C++. With OpenMP it is possible to describe how computations are shared on different threads running on one or different processors or cores. To do so *compiler directives* (often called only directives) are used to specify for the compiler how the instructions have to be computed in parallel. Of course, the compiler has to support OpenMP which is nowadays the case for most compilers (e.g. compilers from GNU, IBM, Intel, PGI, Pathscale etc.).

In many cases it is possible to parallelize sequential code with OpenMP, especially when the code contains loops and the data dependency is low. Sometimes this is not enough to get a sufficient level of performance and a reorganization of the code is needed to create parallelism. One example for such a procedure is the modification of the CG-algorithm to the so called pipelined-CG which is e.g. explained in [**SG06**].

The concept of OpenMP is to execute a program by a collection of cooperating threads. To execute a program the operating system generates a process which consists of several threads. These threads share the resources that have been assigned to the process in the sense that the memory address space is the same for all threads (shared memory). The threads can be processed in parallel on any system offering shared memory, which can be multi-core processors or multiprocessor systems. OpenMP realizes a fork-join programming model, where a master thread starts the program execution. If the execution of the program arrives at a OpenMP parallel pragma, a so called "team of threads" will be created to execute the parallel section. Such a process is called *fork*. At the end of the parallel region only the master thread continues and all other threads are terminated. *Join* is the most common notation for this process. Beside creating teams of threads and specify

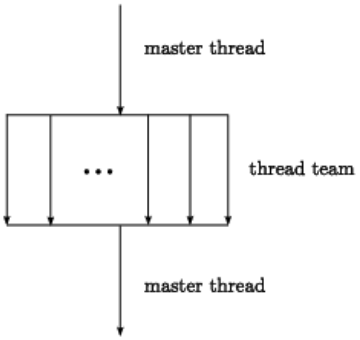


Figure 2.6. "Fork-Join"-model of OpenMP. Multiple threads are created within the execution process of a program when a parallel region is accessed by the master thread ("fork"). When the parallel region is left, only the master thread continues the work and all other threads are terminated ("join"). Communication is performed based on shared variables.

how to share work among this team, OpenMP provides also means for synchronizing threads and declaring thread exclusive data. Synchronization is often needed for communication between several threads which is performed implicitly via shared variables in memory. OpenMP allows an implicit and an explicit work distribution among threads. A loop can be parallelized by placing a pragma before the loop starts and the parallelization is done automatically or in some sense implicit. In contrast to that it is also possible to assign work explicitly to different threads. This is called SPMD (single program multiple data) programming.

OpenMP specifications can be found in [**Ope08**] and detailed description of available directives can be found in [**CJvdP08**], [**HL08**] or [**RR07**].

2.2.2. MPI. MPI (Message Passing Interface) realizes a distributed memory programming model providing Fortran, C and C++ support. It provides a broad set of routines for managing processes and explicit communication. An MPI-program consists of different processes containing private data. Actually every process could execute a different program, what would be called MPMD (*multiple program multiple data*). However, every process often executes the same program, what is called SPMD (*single program multiple data*).

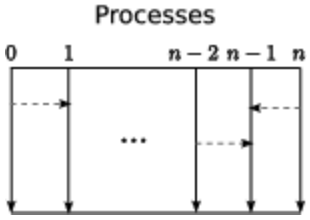


Figure 2.7. Process-based program execution. $n+1$ processes are initialized and they are communicating with each other by explicitly sending messages.

The communication routines are the most important part of MPI. There are point-to-point communication routines like *send* and *receive*, where only two processes are involved. In this case the sender and the receiver have to call the fitting routine. This is the basic MPI communication mechanism. On the other hand MPI provides collective communication routines, where a complete group of processes is involved. These are for example broadcast, gather, scatter, global reduction or barrier (synchronization) routines. All processes which participate in such an collective communication need to be organized in a MPI *communicator*. The standard communicator, in which all processes are included, is *MPI_COMM_WORLD*. The MPI-2 standard also facilitates one-sided communication, where only one process calls a communication routine to manipulate data in the address space of another process. Furthermore communication routines are distinguished between blocking and non-blocking routines. If a blocking communication routine returns, the communication is completed and the resources involved in the communication can be used again. A non-blocking routine may return before the communication is completed, therefore the resources cannot be reused immediately. There exist special routines to check if a non-blocking operation is completed or not.

MPI-1 is based on a static process model, that is processes are created at the start of the program execution. Creating or deleting processes during the program runtime is not possible. MPI-2 takes a dynamic process model as a basis. Thus creating and deleting processes using special routines at runtime is possible.

More information about MPI and a list of MPI routines can be found in [**Mes09**], [**RR07**] or [**ALO02**].

2.2.3. CUDA and OpenCL. Employing graphics processing units (GPU) for general purpose computations (GPGPU) is not new. There are currently three mayor producers of graphics cards, AMD (formerly ATI which was purchased by AMD in November 2006), Intel and NVIDIA. While Intel currently focuses on medium performance on-board graphics and combined CPU-GPU processors, AMD and NVIDIA also offer dedicated hardware and APIs for GPGPU applications. Historically, the strength of GPUs are single precision floating-point operations. New GPU generations can also provide double precision and integer operations (NVIDIA in hardware, AMD emulated), however with reduced FLOP rates compared to single precision.

Choosing the single precision performance for calculating the power efficiency, modern GPUs need about 0.2 W/GFLOPS (Nvidia GTX 480: 250 W, 1345 GFLOPs in single precision). In terms of performance per Watt this are good theoretical operating figures, but more than 200 W power consumption (and related heat that has to be removed from the components) makes those chips hardly eligible for e.g. blade servers. Due to the fact that only a minority of customers of computing

centers (can) use GPUs they are not wide spread. Nvidia and AMD offer two major lines of GPUs. One is dedicated to private consumers (Geforce/Radeon) and one is for so called professional users (Tesla/Firestream). The difference of the two lines is usually the support for the user, quality/reliability of the components, hardware capabilities, the amount of memory and special properties like error correcting memory (ECC).

Programming GPUs can be done with conventional graphics languages like OpenGL, but both NVIDIA and AMD offer more general programming models for their chips too. AMD offered till recently Close to Metal (CTM) which is a quite low level API, that can be used with Brook+, a variation of the higher level GPU programming language Brook developed at the University of Stanford. Brook+ is still near to stream processing (vectors of variable length) and requires major code changes. NVIDIA's solution is called "Compute Unified Device Architecture" (CUDA) [**NVI09**], originally an extension to the C programming language which can nowadays also be used with multiple other languages like Python, Perl, Fortran, Java, Ruby, Lua, and MATLAB (partly by employing third party wrappers). The support of Fortran is based on the fact that still a lot of code, especially in the area of scientific computing, is written in Fortran. As mentioned above, AMD focuses now on OpenCL.

Using CUDA enables developers to access the virtual instruction set and memory of the parallel computational elements in CUDA GPUs and NVIDIA GPUs become accessible for computation like CPUs but in a little less convenient way. GPUs are optimized to execute many concurrent threads not extremely fast while todays CPUs are optimized to execute only a few threads very fast. The last statement seems to change with the emergence of hybrid CPUs combining traditional CPUs with GPUs on one die (e.g. AMD Fusion and the Clarkdale architecture from Intel).

General purpose computation on graphics processing units depicts calculations done on a GPU, whose primary purpose is not video output. As early as 1990, GPUs could be used for robot motion planning, long before 3D capabilities were introduces. Another technique from 2001 renders two textures in an ingenious fashion on a cube, such that the use of particular parameters for perspective and texture overlay yields the matrix product of the textures 8 bit color values. A texture size of 1024×1024 pixels reached in this way more then 4 GBOPS (byte operations per second). Because of this inconvenient programming, the beginning of GPGPU is rather connected to the introduction of the fully programmable graphics pipeline also in 2001.

The problems GPGPU deals with, were originally strongly connected to computer graphic and image processing. With the introduction of full floating-point support, scientific projects started implementing finite difference and finite element techniques for the solution of systems of partial differential equations (PDEs). Several papers in 2003 demonstrated e.g. solutions of the Navier-Stokes equations for incompressible fluid flow on the GPU or for boundary value problems. Meanwhile, even industrial applications using GPUs start emerging.

The Open Computing Language (OpenCL) is a framework for programming heterogeneous platforms consisting of CPUs, GPUs and DSPs. Originally developed by Apple, OpenCL was further developed by multiple industrial companies (AMD, Intel, IBM, Nvidia) and in december 2008 as standard released. From a architectonic point of view a OpenCL device consists of a *host* and one or more independent *compute units*. A compute unit can consist of one or more *compute units*. One major task of the host is to distribute the kernels onto the available compute units during runtime. OpenCL kernels are translated by the OpenCL-compiler during runtime and directly executed on the OpenCL device. One significant advantage

of OpenCL is that it is much more platform independent then CUDA which works only on Nvidia GPUs.

2.2.4. PGAS Languages. Within this thesis, PGAS (Partitioned Global Address Space) Languages are not taken into account, however they shall be characterized. This section is, exept for minor changes, taken from [**BBC+08**] (page 45).

PGAS combines some of the features of message passing and shared memory threads. Like a shared memory model, there are shared variables including arrays and pointer-based structures that live in a common address space, and are accessible to all processes. But like message passing, there the address space is logically "partitioned" so that a particular section of memory is viewed as "closer" to one or more processes. In this way the PGAS languages provide the needed locality information to map data structure efficiently and scalable onto both shared and distributed memory hardware. The partitioning provides different execution and performance-related characteristics, namely fast access through conventional pointers or array indexes to nearby memory and slower access through global pointers and arrays to data that is far away. Since an individual process may directly read and write memory that is near another process, the global address space model directly supports one-sided communication: no participation from a remote process is required for communication. Because PGAS languages have characteristics of both shared memory threads and (separate memory) processes, some PGAS languages use the term "thread" while others use "process". The model is distinguishable from shared memory threads such as OpenMP, because the logical partitioning of memory gives programmers control over data layout. Arrays may be distributed at creation time to match the access patterns that will arise later and more complex pointer-based structures may be constructed by allocating parts in each of the memory partitions and linking them together with pointers.

The PGAS model is realized in three decade-old languages, each presented as an extension to a familiar base language: United Parallel C (UPC) [**Con05**] for C; Co-Array Fortran (CAF) [**NR98**] for Fortran, and Titanium [**YSP+98**] for Java. The three PGAS languages make references to shared memory explicit in the type system, which means that a pointer or reference to shared memory has a type that is distinct from references to local memory. These mechanisms differ across the languages in subtle ways, but in all three cases the ability to statically separate local and global references has proven important in performance tuning. On machines lacking hardware support for global memory, a global pointer encodes a node identifier along with a memory address, and when the pointer is dereferenced, the runtime must deconstruct this pointer representation and test whether the node is the local one. This overhead is significant for local references, and is avoided in all three languages by having expression that are statically known to be local, which allows the compiler to generate code that uses a simpler (address-only) representation and avoids the test on dereference.

These three PGAS languages used a static number of processes fixed at job start time, with identifiers for each process. This Single Program Multiple Data (SPMD) model results in a one-to- one mapping between processes and memory partitions and allows for very simple runtime support, since the runtime has only a fixed number of processes to manage and these typically correspond to the underlying hardware processors. The languages run on shared memory hardware, distributed memory clusters, and hybrid architectures. On shared memory systems and nodes within a hybrid system, they typically use a thread model such as Pthreads for the underlying execution model. The distributed array support in all three languages

is fairly rigid, a reaction to the implementation challenges that plagued the High Performance Fortran (HPF) effort. In UPC distributed arrays may be blocked, but there is only a single blocking factor that must be a compile-time constant; in CAF the blocking factors appear in separate "co-dimensions;" Titanium does not have built-in support for distributed arrays, but they are programmed in libraries and applications using global pointers and a built-in all-to-all operation for exchanging pointers. There is an ongoing tension in this area of language design between the generality of distributed array support and the desire to avoid significant runtime overhead.

Each of the languages is also influenced by the philosophy of their base serial language. Co- Array Fortran support is focused on distributed arrays, while UPC and Titanium have extensive support for pointer-based structures, although Titanium also breaks from Java by adding extensive support for multidimensional arrays. UPC allows programmers to deconstruct global pointers and to perform pointer arithmetic such a incrementing pointers and dereferencing the results. Titanium programs retain the strong typing features of Java and adds language and compiler analysis to prove deadlock freedom on global synchronization mechanisms.

CHAPTER 3

Multiprecision Methods

As first part in this chapter, the iterative refinement method for solving linear systems of equations is explained. In addition to the principle method, the utilization of mixed precision techniques is considered as well as convergence of such techniques. For the special case of elliptic operators, analytical results are shown regarding the perturbation that can be applied to the arising linear systems within the solution process when multiple floating point formats are utilized.

Parts of this chapter have already been presented and published in the context of the International Conference on State of the Art in Scientific and Parallel Computing (PARA 2010) and the International Meeting on High Performance Computing for Computational Science (VECPAR 2010). Related publications are [**AHR**] and [**AHR11**].

3.1. Iterative Refinement Method

The motivation for the iterative refinement method can be obtained from Newton's method. Here, f is a given function and x_i represents the solution approximation in the ith step:

$$\begin{aligned} x_0 \quad & \text{given} \\ x_{i+1} &= x_i - (\nabla f(x_i))^{-1} f(x_i), \quad i = 0, 1, 2, \dots \quad . \end{aligned} \tag{3.1}$$

This method can be applied to the function $f(x) = b - Ax$ with $\nabla f(x) = -A$, where $Ax = b$ is the linear system that should be solved with $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$.

By defining the residual according to $r_i := b - Ax_i$, one obtains

$$\begin{aligned} x_{i+1} &= x_i - (\nabla f(x_i))^{-1} f(x_i) \\ &= x_i + A^{-1}(b - Ax_i) \\ &= x_i + A^{-1} r_i \quad i = 0, 1, 2, \dots \quad . \end{aligned}$$

Denoting the approximation for the solution update with $\tilde{c}_i \approx A^{-1} r_i$ and using an initial guess x_0 as starting value, an iterative algorithm can be defined, where any linear solver can be used as error correction solver.

Algorithm 17 Iterative Refinement Method

1: initial guess as starting vector: x_0
2: compute initial residual: $r_0 = b - Ax_0$
3: **while** ($\| Ax_i - b \| > \varepsilon \| r_0 \|$) **do**
4: $\quad r_i = b - Ax_i$
5: $\quad$ solve error correction equation: $Ac_i = r_i$ approximately by $\tilde{c}_i$
6: $\quad$ update solution: $x_{i+1} = x_i + \tilde{c}_i$

In each iteration, the error correction solver searches for a $\tilde{c}_i$ such that $A\tilde{c}_i \approx r_i$ (with respect to a certain quality, see. e.g. Chapter 1.6). Afterwards, the solution

approximation is updated by $x_{i+1} = x_i + \tilde{c}_i$ until the outer residual stopping criterion with a given $0 < \varepsilon < 1$ is fulfilled.

3.1.1. Error Correction Solver. Due to the fact that the iterative refinement method makes no demands on the inner error correction solver, any backward stable linear solver can be chosen. Still, especially the Krylov subspace methods have turned out to be an adequate choice for many cases. These provide an approximation of the residual error iteratively in every computation loop, which can efficiently be used to control the stopping criterion of the error correction solver (cf. Chapter 1). The Krylov subspace methods used in the presented experiments (see Chapter 3.2.3) fulfill the demand of backward stability, [**DRDSA**$^+$**95**] and [**BES98**].

3.1.2. Convergence Analysis of Iterative Refinement Methods. It should be pointed out that the below presented results do not include rounding errors and assume convergence of the error correction solver.

Based on the residual $r_i = b - Ax_i$ in the ith step, the improvement (w.r.t. accuracy) associated with one iteration loop of the iterative refinement method can be analyzed. This is done to derive an estimation for the number of outer iterations needed to achieve a certain accuracy.

Applying a solver to the error correction equation $Ac_i = r_i$ generates a solution approximation $\tilde{c}_i$. The associated residual d_i of the error correction solver is defined according to

$$d_i := r_i - A\tilde{c}_i, \quad i = 0, 1, 2, \dots \ .$$

In case of a Krylov subspace method as inner solver, a residual stopping criterion can be taken (see Chapter 1.6 for more information) which means that the residual of the error correction solver fulfills:

$$\| d_i \| \leq \varepsilon_{inner} \| r_i \| \quad i = 0, 1, 2, \dots \ . \tag{3.2}$$

After updating the solution like $x_{i+1} = x_i + \tilde{c}_i$, the new residual error term can be obtained:

$$\begin{aligned} \| r_{i+1} \| &= \| b - Ax_{i+1} \| \\ &= \| b - A(x_i + \tilde{c}_i) \| \\ &= \| \underbrace{b - Ax_i}_{=r_i} \underbrace{-A\tilde{c}_i}_{=d_i - r_i} \| \\ &= \| d_i \| \overset{(3.2)}{\leq} \varepsilon_{inner} \| r_i \|, \quad i = 0, 1, 2, \dots \ . \end{aligned}$$

Hence, the accuracy improvements obtained by performing one iteration loop equal the accuracy of the residual stopping criterion of the error correction solver. It follows, that after i iteration loops, the residual r_i fulfills

$$\| r_i \| \leq \varepsilon_{inner}^i \| r_0 \|, \quad i = 0, 1, 2, \dots \ . \tag{3.3}$$

If i is choosen such that

$$\varepsilon_{inner}^i \|r_0\| \leq \varepsilon \|r_0\|, \tag{3.4}$$

then it follows from 3.3 that

$$\|r_i\| \leq \varepsilon \|r_0\|.$$

If we assume $0 < \varepsilon_{inner} < 1$ then 3.4 holds if

$$i \geq \frac{\log(\varepsilon)}{\log(\varepsilon_{inner})}. \tag{3.5}$$

Hence the number of outer iterations which guarantee $\|r_i\| \leq \varepsilon \|r_0\|$ is

$$i = \left\lceil \frac{\log(\varepsilon)}{\log(\varepsilon_{inner})} \right\rceil . \tag{3.6}$$

where $\lceil \cdot \rceil$ denotes the Gaussian ceiling function.

Results of numerical experiments validating the theoretical findings can be found in Chapter 3.2.3.

3.2. Mixed Precision Iterative Refinement Solvers

3.2.1. Mixed Precision Approach. The underlying idea of mixed precision iterative refinement methods is to use different precision formats within the algorithm of the iterative refinement method, approximating the relative residual error and updating the solution approximation in high precision, but computing the error correction term in a lower precision format (see Figure 3.1). This approach was also suggested by [**BBD$^+$08**], [**BDK$^+$08**], [**BDL$^+$07**], [**GST07**] and [**GS08**].

Using the mixed precision approach to the iterative refinement method, users have to be aware of the fact that the residual error bound of the error correction solver may not exceed the accuracy of the lower precision format.

Furthermore, each error correction produced by the inner solver in lower precision cannot exceed the data range of the lower precision format. This means that the smallest possible error correction is the smallest number ϵ_{low}, that can be represented in the lower precision. Thus, the accuracy of the final solution cannot exceed ϵ_{low} either. This can become a problem when working with very small numbers, because then the solution correction terms can not be denoted in low precision, but in most cases, the problem can be avoided by converting the original values to a higher order of magnitude. Instead of solving the error correction equation $Ac_i = r_i$, one applies the error correction solver to the system $Ac_i = 10^p r_i$ where p has to be chosen such that the solution update c_i can be represented in the used low precision format. In this case, the solution update in high precision becomes $x_{i+1} = x_i + 10^{-p} c_i$.

But there are also some more demands to the used low precision floating point format and the used error correction solver with its respectively stopping criterion. While the low precision floating point format has to be chosen with respect to the condition number of the linear system such that the linear system is still solvable within this format, it has to be ensured that the used error correction solver in low precision converges for the given problem, and does not stagnate before the demanded accuracy of the solution update is achieved. At this point it should be mentioned, that the condition number of the low precision representation of the matrix A may differ from the condition number of the original system.

If the final accuracy does not exceed the smallest number that can be represented in the lower precision format, and if the condition number of the linear system is small enough such that the system is solvable in low precision and the used error correction solver converges and does not stagnate before the demanded accuracy is achieved, then the mixed precision iterative refinement method gives (in terms of the stopping criteria) the same solution approximation as if the solver was performed in the high precision format. It is important to denote that the results are not bit-wise the same!

When comparing the algorithm of an iterative refinement solver using a certain Krylov subspace solver as error correction solver to the plain Krylov solver, it can

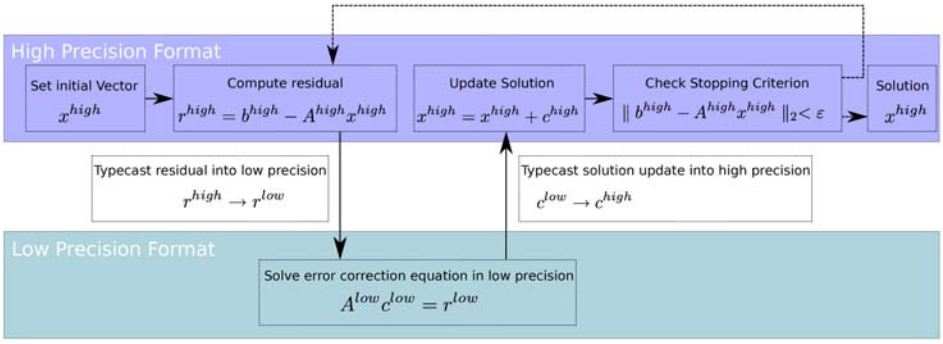


Figure 3.1. Mixed precision approach applied to an iterative refinement solver.

be realized, that the iterative refinement method has more computations to execute due to the additional residual computations, solution updates and typecasts.

Each outer loop consists of the computation of the residual error term, a typecast, an initialization of a vector, the scaling process, the inner solver for the correction term, the reconversion of the data and the solution update. The computation of the residual error itself consists of a matrix-vector multiplication, a vector addition and a scalar product. Using a hybrid architecture, the converted data often has to be transmitted between the devices.

For the typecast of a vector of the dimension n, the same cost as for one vector addition in the higher precision format is assumed. For the matrix-vector multiplication the cost of $2 \cdot nnz$ are assumed, where nnz is the number of nonzero entries within the matrix.

When using hybrid hardware with a distributed memory address space, the data transfer can be displayed with respect to the cost of one computation in high precision as $\alpha + \beta \cdot n$, where α denotes the delay (or latency) and β the effort needed to transmit one vector entry. nnz denotes the number of nonzero entries of the matrix. Altogether, for dimension n this gives an additional complexity to the plain solver of:

additional operations	additional cost
matrix vector multiplication	$2 \cdot nnz$
2 vector additions	$2 \cdot n$
2 vector scalings	$2 \cdot n$
2 vector typecasts	$2 \cdot n$
vector initialization	n
2 scalar products	$2 \cdot (2n - 1)$
2 vector transmissions	$2 \cdot (\alpha + \beta n)$
$\sum$	$2 \cdot nnz + 11n - 2 + 2\alpha + 2\beta n$

The goal is to analyze in which cases the mixed precision iterative refinement method outperforms the plain solver in high precision. Obviously this is the case if the additional operations (denoted with K) are overcompensated by the cheaper execution of the iterative error correction solver in low precision. Using an explicit residual computation the computational cost of K is in the magnitude of the matrix-vector multiplication. In case of an iterative update for the residual, the complexity is even lower.

3.2.2. Convergence Analysis of Mixed Precision Approaches. When discussing the convergence of the iterative refinement method in section 3.1.2, a model for the number of outer iterations needed to obtain a residual error below a certain residual threshold $\varepsilon \parallel r_0 \parallel$ was derived. Having a relative residual stopping criterion ε_{inner} of the Krylov subspace solver used as error correction solver

$$i = \left\lceil \frac{\log(\varepsilon)}{\log(\varepsilon_{inner})} \right\rceil$$

iterations have to be performed (according to **??**) to obtain an approximation x_i which fulfills

$$\parallel r_i \parallel = \parallel b - Ax_i \parallel \leq \varepsilon \parallel b - Ax_0 \parallel = \varepsilon \parallel r_0 \parallel \quad i = 1, 2, 3, \ldots \ .$$

It should be mentioned, that this stopping criterion can only be fulfilled, if the Krylov subspace solver converges in the respectively used floating point format.

If the iterative refinement technique in mixed precision is used, this convergence analysis has to be modified due to the floating point arithmetic. In fact, two phenomena may occur that require additional outer iterations:

(1) Independently of the type of the inner error correction solver, the low precision format representations of the matrix A and the residual r_i contain representation errors due to the floating point format. These rounding errors imply that the error correction solver performs the solving process to a perturbed system $(A + \delta A)c_i = r_i + \delta r_i$. Due to this fact, the solution update c_i gives less improvement to the outer solution than expected. Hence, the convergence analysis of the iterative refinement method has to be modified when using different precision formats. To compensate the smaller improvements to the outer solution, additional outer iterations have to be performed.
(2) When using a Krylov subspace method as inner correction solver, the residual is computed iteratively within the solving process. As floating point formats have limited accuracy, the iteratively computed residuals may differ from the explicit residuals due to rounding errors. This can lead to an early breakdown of the error correction solver. As in this case the improvement to the outer solution approximation is smaller than expected, the convergence analysis for iterative refinement methods using Krylov subspace solvers as error correction solvers has to be modified furthermore. It may happen, that additional outer iterations are required to compensate the early breakdowns of the error correction solver.

By denoting the total number of additional outer iterations induced by the rounding errors and the early breakdowns when using Krylov subspace methods for the inner solver with g one obtains

$$i_{total} = \left\lceil \frac{\log \varepsilon}{\log \varepsilon_{inner}} \right\rceil + g \tag{3.7}$$

for the total number of outer iterations. In fact g does not only depend on the type of the error correction solver, but also on the used floating point formats, the conversion and the properties of the linear problem including the matrix structure.

In order to be able to compare a mixed precision iterative refinement solver to a plain high precision solver, a model serving can be derived as an upper bound for the computational cost. The complexity of a Krylov subspace solver generating a solution approximation with the relative residual error ε is denoted as $C_{solver}(\tilde{\varepsilon})$. This complexity estimation can be obtained from the convergence analysis of the Krylov subspace solvers [**Saa03**]. Using this notation, the complexity $C_{mixed}(\varepsilon)$ of

an iterative refinement method using a correction solver with relative residual error ε_{inner} can be displayed as

$$C_{mixed}(\varepsilon) = \left(\left\lceil \frac{\log(\varepsilon)}{\log(\varepsilon_{inner})} \right\rceil + g \right) \cdot \left(C_{solver}(\varepsilon_{inner}) \cdot s + K \right), \tag{3.8}$$

where $s \leq 1$ denotes the acceleration gained by performing computations in the low precision format (eventually parallel on the low precision device) instead of the high precision format. The quotient between the mixed precision iterative refinement approach to a certain solver and the plain solver in high precision is denoted with $f_{solver} = \frac{C_{mixed}(\varepsilon)}{C_{solver}(\varepsilon)}$, and

$$f_{solver} = \frac{\left(\left\lceil \frac{\log(\varepsilon)}{\log(\varepsilon_{inner})} \right\rceil + g \right) \cdot \left(C_{solver}(\varepsilon_{inner}) \cdot s + K \right)}{C_{solver}(\varepsilon).} \tag{3.9}$$

can be obtained.

By analyzing this fraction, the following propositions can be stated:

(1) If $f_{solver} < 1$, the mixed precision iterative refinement approach to a certain solver performs faster than the plain precision solver. This superiority of the mixed precision approach will particularly occur, if the speedup gained by performing the inner solver in a lower precision format (e.g. on a accelerator) overcompensates the additional computations, typecasts and the eventually needed transmissions in the mixed precision iterative refinement method.
(2) The inverse $\frac{1}{f_{solver}}$ could be interpreted as *speedup factor* obtained by the implementation of the mixed precision refinement method with a certain error correction solver. Although this notation does not conform with the classical definition of the speedup concerning the quotient of a sequentially and a parallel executed algorithm, $\frac{1}{f_{solver}}$ can be construed as measure for the acceleration triggered by the use of the mixed precision approach (and the eventually hybrid system).
(3) The iteration loops of Krylov subspace solvers are usually dominated by a matrix-vector multiplication. Hence, using a Krylov subspace method as inner error correction solver, the factor f_{solver} is independent of the problem size for large dimension. This can also be observed in numerical experiments (see Chapter 3.2.3 and [**ABV10**]).

Exact knowledge of all parameters would enable to determine a priori whether the mixed precision refinement method using a certain error correction solver outperforms the plain solver. The computational cost of a Krylov subspace solver depends on the dimension and the condition number of the linear system [**Saa03**].

While the problem size can easily be determined, an approximation of the condition number of a certain linear system can be obtained by performing a certain number of iterations of the plain Krylov subspace solver, and analyzing the residual error improvement. Alternative methods to obtain condition number estimations can for example be found in [**Hig96**].

The only factor that poses problems is g, the number of additional outer iterations required to correct the rounding errors generated by the use of a lower precision format for the inner solver. As long as users do not have an estimation of g for a certain problem, it is not possible to determine a priori, which solver performs faster.

To resolve this problem, an implementation of an intelligent solver suite could use the idea to determine a posterior an approximation of g, and then choose the optimal

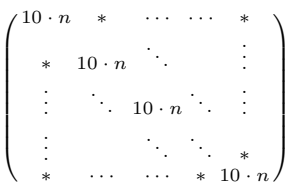
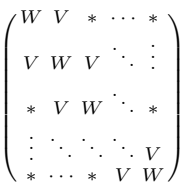
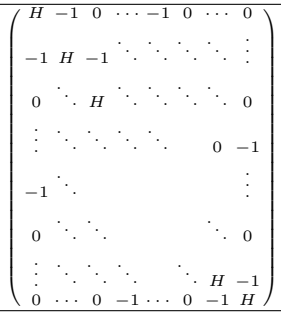
solver. To get an a posterior approximation of g, the solver executes the first iteration loop of the inner solver and then compares the improvement of the residual error with the expected improvement. Through the difference, an estimation for the number of additional outer iterations can be obtained, that then enables to determine the factor f_{solver} and choose the optimal version of the solver.

3.2.3. Numerical Experiments. In this section three facts are shown based on a set of experiments:

(1) Depending on the condition number of the system, the plain solver or the mixed precision iterative refinement variant is superior.
(2) The factor f_{solver} is independent of the dimension of the problem. This includes, that the mixed precision method works better for reasonably many problems.
(3) The total number of outer iterations i_{total} (3.7) using limited precision usually differs only by a small value g from the theoretical value i (**??**). Hence approximating i_{total} by $i + 2$ is usually a reasonable estimation.

In order to show these results, a set of artificially created test-matrices is used with fixed condition number but increasing dimension.

To the linear system affiliated to these matrices, a CG solver as well as a GMRES solver are applied and the performance is compared to the respective mixed precision implementations. All solvers use the relative residual stopping criterion $\varepsilon = 10^{-10} \parallel r_0 \parallel_2$. As right hand side a vector with ones in each component and the zero-vector as initial guess for the solution has been chosen (aware of the fact that a multiple of the right hand side leads often to reasonable results and that there exist a large variety of application dependent strategies for choosing good initial guess like taking the solution of the last time-step etc.). Due to the iterative residual computation in the case of the plain solvers, the mixed precision iterative refinement variants usually iterate to a better approximation, since they compute the residual error explicitly, but as the difference is generally small, the solvers are comparable. For the mixed precision iterative refinement implementations $\varepsilon_{inner} = 10^{-1}$ is used. The GMRES algorithm, as explained in Algorithm 11, is equipped with a restart parameter of 10.

M1	M2	M3
		
$* = rand(0,1)$	$* = rand(0,1)$ $V = 10^3 \cdot n$ $W = 2 \cdot 10^3 \cdot n + n$	$H = 4 + 10^{-3}$
problem: artificial problem size: variable sparsity: $nnz = n^2$ cond. num.: $\kappa < 3$ storage format: MAS	problem: artificial problem size: variable sparsity: $nnz = n^2$ cond. num.: $\kappa \approx 8 \cdot 10^3$ storage format: MAS	problem: artificial problem size: variable sparsity: $nnz \approx 5n$ cond. num.: $\kappa \approx 8 \cdot 10^3$ storage format: CRS

Tab. 1: Structure plots and properties of the artificial test-matrices.

A more detailed description of the used test matrices, and a more extensive set of numerical experiments including physical applications, can be found in Chapter 5.4.

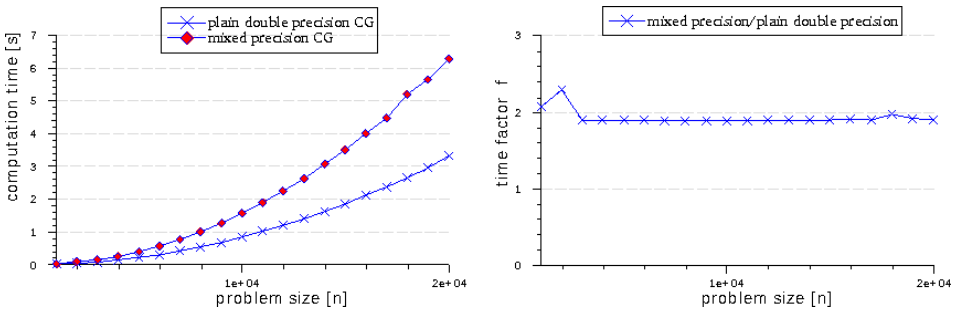


Figure 3.2. Performance of CG/mixed CG applied to test case M1

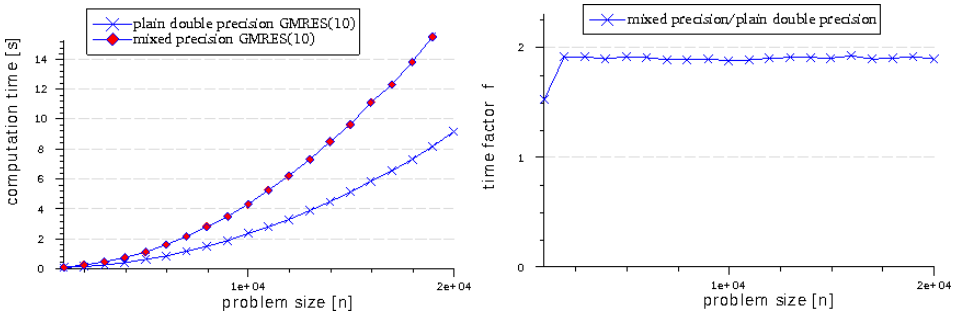


Figure 3.3. Performance of GMRES/mixed GMRES applied to test case M1

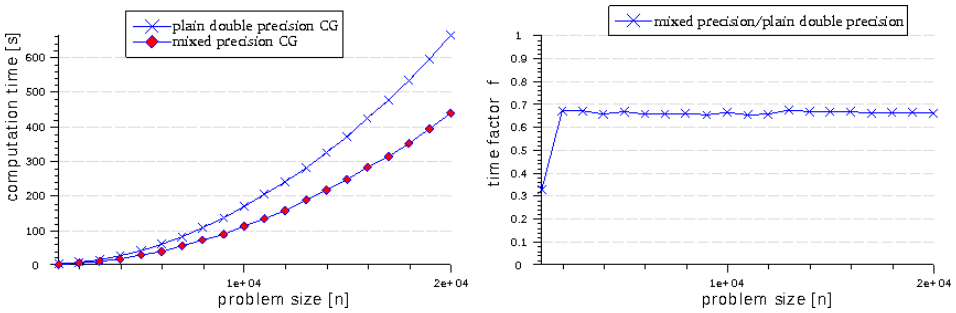


Figure 3.4. Performance of CG/mixed CG applied to test case M2

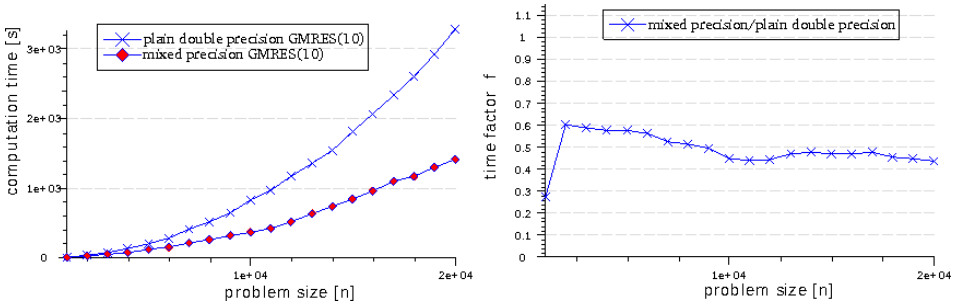


Figure 3.5. Performance of GMRES/mixed GMRES applied to test case M2

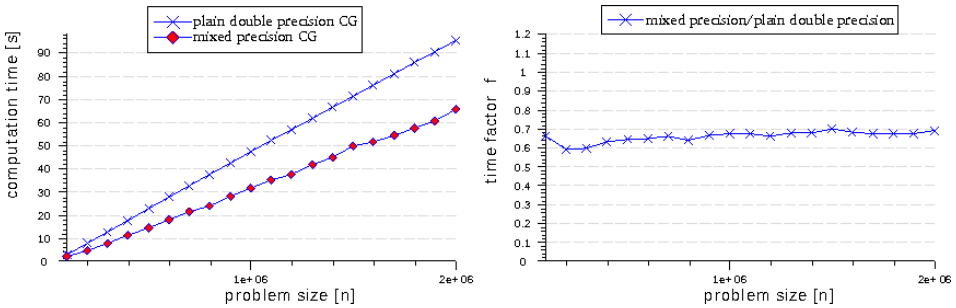


Figure 3.6. Performance of CG/mixed CG applied to test case M3

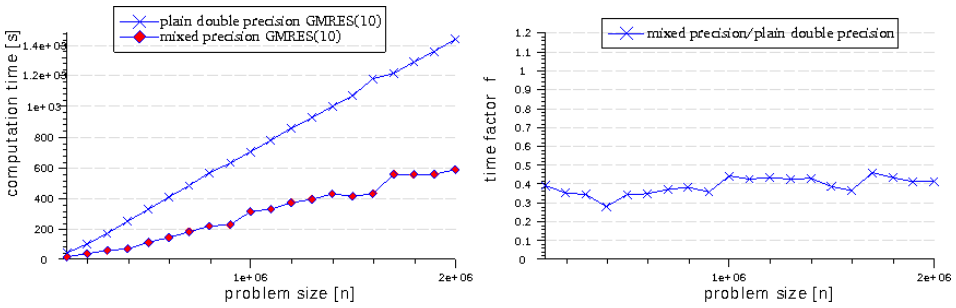


Figure 3.7. Performance of GMRES/mixed GMRES applied to test case M3

The previous sections showed results of numerical analysis concerning the convergence theory of mixed precision iterative refinement methods. These results from the current section contribute to the possibility to control the usage of different precision formats within an error correction solver.

A problem still requiring a more satisfactory solution is to determine the exact dependency of the number of additional outer iterations on the characteristics of the linear system, the solver type, the inner and outer stopping criterion, and the used floating point precision formats. Further work in this field is needed to enable an estimation depending on these parameters.

Technologies like FPGAs and application-specific designed processors offer a free choice of floating point formats. Controlling the usage of these precision formats within iterative refinement solvers is essential for optimizing the performance.

3.3. Mixed Precision Applied to Elliptic Operators

This chapter includes an analysis of the error propagation within a mixed precision iterative refinement solver as explained in Chapter 3.1. The aim is to derive bounds for the complexity of the low precision format used for the inner error correction solver that guarantee the convergence of the error correction method. The analysis reveals that these bounds depend on the discretization. As long as the rounding error induced by the limited precision is not of higher order than the discretization error, the error estimations only differ by a constant depending on the floating point format.

3.3.1. Motivation. When within the iterative refinement method (see Chapter 3.1) mixed precision techniques (see Chapter 3.2.1) are utilized in order to accelerate the solver the question arises, how far the complexity of the floating point format can be modified without worsening the quality of the result.

For elliptic equations, there exists the lemma of Strang [**Bra07**], providing an estimation for the error when discretizing the operator with finite elements. This estimation can be used to derive an upper bound for the complexity of the floating point format which can be used within the solution process when mixed precision approaches are utilized.

The underlying idea is to use the estimation

$$\|u - u_h\| \leq ch^2\|f\|,$$

where u_h denotes an approximation to the exact solution u of an elliptic equation, h is the discretization fineness and c a problem specific constant. This estimation can be obtained from the lemma of Aubin-Nitsche for special discretizations consisting of quasi uniform triangles [**Bra07**].

In the end, it is obtained that as long as the rounding error triggered by the limited accuracy of the used floating point format is not of higher order than the error triggered by the discretization of the operator, the error stays in the same order of magnitude. In this case, the regularity of the obtained linear system is preserved.

3.3.2. Numerical Analysis of Perturbations for Elliptic Operators. At first the general relation between discretization and method error is analyzed. Assuming that the problem is to search $u \in V$ based on a variational problem of the form

$$a(u, v) = \langle f, v \rangle \quad \forall v \in V \tag{3.10}$$

where $V \subset H^m(\Omega)$ and $a : V \times V \to \mathbb{R}$ is a symmetric, elliptic and positive bilinear form, which means that $a(u,u) > 0$ for all $u \in V, u \neq 0$. $\Omega \subset \mathbb{R}$ is assumed to be bounded and $f \in L_2(\Omega)$.

Now a finite dimensional discretization of $a(\cdot,\cdot)$ and f is considered based on a discretization width h and search for an approximation $u_h \in V_h$ such that

$$a_h(u_h, v_h) = \langle f_h, v_h \rangle \quad \forall v_h \in V_h. \tag{3.11}$$

It is assumed that the discretized bilinear form $a_h(\cdot,\cdot)$ is uniformly elliptic, which means there exists a constant $\alpha > 0$ independent of the discretization such that

$$a_h(v_h, v_h) \geq \alpha \|v_h\|^2_{m,\Omega} \quad \forall v_h \in V_h. \tag{3.12}$$

Then the Lemma of Strang can be applied providing the estimation

$$\begin{aligned} \|u - u_h\|_m \leq c_s \Big[& \inf_{v_h \in V_h} \Big(\|u - v_h\|_m && \text{(Approximation error)} \\ & + \sup_{w_h \in V_h} \tfrac{|a(v_h,w_h) - a_h(v_h,w_h)|}{\|w_h\|_m} \Big) && \text{(Consistency error of } a\text{)} \\ & + \sup_{w_h \in V_h} \tfrac{\langle f, w_h \rangle - \langle f_h, w_h \rangle}{\|w_h\|_m} \Big] && \text{(Consistency error of } f\text{)} \end{aligned}$$

with a constant c_s independent of the discretization.

In the following the continuity constant is needed and now defined. By assuming a Hilbert space H, a bilinear form $a : H \times H \to \mathbb{R}$ is called continuous if there exists a $C > 0$ fulfilling

$$|a(u,v)| \leq C \|u\| \cdot \|v\| \quad \forall u, v \in H. \tag{3.13}$$

If it is furthermore assumed that the Hilbert space $V \subset H^m$ is a continuous embedding, the validity of the Aubin-Nitsche lemma holds [**Bra07**]. It states, that the finite element solution u_h in $V_h \subset V$ satisfies the estimation

$$\|u - u_h\|_0 \leq c_1 \|u - u_h\|_1 \sup_{g \in H} \left(\frac{1}{\|g\|_0} \inf_{v \in V_h} \|\varphi_g - v\|_1 \right)$$

where for every $g \in H$, $\varphi_g \in V$ denotes the corresponding unique weak solution of the equation

$$a(w, \varphi_g) = \langle g, w \rangle \quad \forall w \in V.$$

The detailed proof of this result can be found in [**Bra07**].

From this, it can be obtained that for many conforming (quasi-uniform) as well as some nonconforming (Crouziex-Raviart) Finite-Element discretization methods, the estimation

$$\|u - u_h\|_0 \leq c_a h^2 \|f\|_0 \tag{3.14}$$

holds, where c_a is a constant independent of the discretization size h [**Bra07**].

Due to the fact that the floating point representation on computers is finite, errors occur when a_h is computed. The representation of a_h in a certain floating point format is denoted as $\tilde{a}_h$ (e.g. double precision). Applying the mixed precision approach to the iterative refinement method, a representation $\tilde{a}_h$ is used for the part computed in high precision. A second representation of a_h in the lower precision part (e.g. single precision), where the error correction term is computed, is needed. This perturbed bilinear operator can be represented as

$$\tilde{\tilde{a}}_h(\cdot,\cdot) := \tilde{a}_h(\cdot,\cdot) + b_1$$

with a perturbation b_1 triggered by the use of the floating point format in the error correction equation. Similarly, the representation of the right hand side f of equation (3.10) is derived in the high and low precision formats

$$\tilde{\tilde{f}}_h := \tilde{f}_h + b_2.$$

Considering the right-hand side of the estimation provided by Strang and information can be obtained about whether equation

$$c_s \left(\inf_{v_h \in V_h} \left(\|u - v_h\|_m + \sup_{w_h \in V_h} \frac{|a(v_h, w_h) - \tilde{\tilde{a}}_h(v_h, w_h)|}{\|w_h\|_m} \right) + \sup_{w_h \in V_h} \frac{|\langle f, w_h\rangle - \langle \tilde{\tilde{f}}_h, w_h\rangle|}{\|w_h\|_m} \right) \leq c_a h^2 \|f\|_0 \tag{3.15}$$

is fulfilled. The existence of a fixed $v_h \in V_h$ is now assumed fulfilling

$$c_s \left(\|u - v_h\|_m + \sup_{w_h \in V_h} \frac{|a(v_h, w_h) - \tilde{\tilde{a}}_h(v_h, w_h)|}{\|w_h\|_m} + \sup_{w_h \in V_h} \frac{|\langle f, w_h\rangle - \langle \tilde{\tilde{f}}_h, w_h\rangle|}{\|w_h\|_m} \right) \leq c_a h^2 \|f\|_0. \tag{3.16}$$

Now the goal is to obtain information about the perturbation. Therefore, the terms

$$\sup_{w_h \in V_h} \frac{|a(v_h, w_h) - \tilde{\tilde{a}}_h(v_h, w_h)|}{\|w_h\|_m} \leq \sup_{w_h \in V_h} \frac{|a(v_h, w_h) - \tilde{a}_h(v_h, w_h)|}{\|w_h\|_m} + \sup_{w_h \in V_h} \frac{|b_1|}{\|w_h\|_m}, \tag{3.17}$$

and

$$\sup_{w_h \in V_h} \left(\frac{|\langle f, w_h\rangle - \langle \tilde{\tilde{f}}_h, w_h\rangle|}{\|w_h\|_m} \right) \leq \sup_{w_h \in V_h} \left(\frac{|\langle f, w_h\rangle - \langle \tilde{f}_h, w_h\rangle|}{\|w_h\|_m} \right) + \sup_{w_h \in V_h} \left(\frac{|b_2|}{\|w_h\|_m} \right) \tag{3.18}$$

are considered. Now it can be derived by combining (3.16), (3.17) and (3.18) that the condition to estimate the error bound is

$$\left(\underbrace{\|u - v_h\|_m}_{\in O(h^2)} + \underbrace{\sup_{w_h \in V_h} \frac{|a(v_h, w_h) - \tilde{a}_h(v_h, w_h)|}{\|w_h\|_m}}_{\in O(h^2)} \right) + \underbrace{\sup_{w_h \in V_h} \left(\frac{|\langle f, w_h\rangle - \langle \tilde{f}_h, w_h\rangle|}{\|w_h\|_m} \right)}_{\in O(h^2)} + \underbrace{\sup_{w_h \in V_h} \left(\frac{|b_1|}{\|w_h\|_m} \right)}_{\overset{!}{\in} O(h^2)} + \underbrace{\sup_{w_h \in V_h} \left(\frac{|b_2|}{\|w_h\|_m} \right)}_{\overset{!}{\in} O(h^2)} \leq \frac{c_a}{c_s} h^2 \|f\|_0 \tag{3.19}$$

If this equation is fulfilled for the perturbations b_i triggered by the limited precision format is used, the method error is not of higher order than the discretization error.

3.3.3. Constants in Detail. For conforming elements the lemma of Céa [**Bra07, Gro07**] can be taken to derive estimations for the perturbation. But in the case of non-conforming elements a more general formulation is needed which can be obtained by using the lemma of Strang and lemma of Aubin-Nitsche, where the consistency error for a and f is also taken into account.

The right-hand side of equation (3.19) can be quantified to find the bounds for the floating point formats that can be used within the solution process. Beside the discretization size h, the norm $\|f\|_0$ has to be computed. Finally, both constants c_s and c_a have to be explained.

c_s arises in the context of the proof of the lemma of Strang: Let $v_h \in S_h$ be arbitrary and use the abbreviation $u_h - v_h = w_h$. Hereby denotes $\|\cdot\|$ an H^m-

or grid-dependent norm. Based on (3.11) and (3.12) and the fact that u is equi-continuous, one gets

$$\begin{aligned}\alpha\|u_h - v_h\|^2 &\leq a_h(u_h - v_h, u_h - v_h) = a_h(u_h - v_h, w_h)\\ &= a(u - v_h, w_h) + [a(v_h, w_h) - a_h(v_h, w_h)] + [a_h(u_h, w_h) - a(u, w_h)]\\ &= a(u - v_h, w_h) + [a(v_h, w_h) - a_h(v_h, w_h)] - [\langle l, w_h\rangle - \langle l, w_h\rangle]\end{aligned}$$

By deviding by $\|u_h - v_h\| = \|w_h\|$ and the continuity of $a(\cdot,\cdot)$, one obtains:

$$\|u_h - v_h\| \leq C\left(\|u - v_h\| + \frac{\|a(v_h, w_h) - a_h(v_h, w_h)|}{\|w_h\|} + \frac{|\langle l, w_h\rangle - \langle l_h, w_h\rangle|}{\|w_h\|}\right)$$

By applying the triangular inequality

$$\|u - u_h\| \leq \|u - v_h\| + \|u_h - v_h\|$$

one can derive

$$\|u - u_h\| \leq \underbrace{(1 + C)}_{=:c_s}\left(\|u - v_h\| + \frac{\|a(v_h, w_h) - a_h(v_h, w_h)|}{\|w_h\|} + \frac{|\langle l, w_h\rangle - \langle l_h, w_h\rangle|}{\|w_h\|}\right).$$

So the constant c_s is defined by $1+C$, where C is the continuity constant. Although more strict estimations for c_s are possible, the above derived approximation is used due to simplicity.

One conclusion from the Aubin-Nitsche lemma (see e.g. [**Bra07, Gro07**]) is

$$\|u - u_h\|_0 \leq \underbrace{c^2 C}_{=:c_a} h^2 \|f\|_0$$

where $u_h \in V_h$ solves the variational problem $a(u_h, v_h) = \langle f, v_h\rangle$ based on linear (respectively quadratic or cubic) triangular elements. Herby denotes C again the continuity constant and it is possible to get the remaining components of c_s as $c := ((1 + c_1)c_2 C)/\alpha$.

From the definition of the H^2-regularity [**Bra07, Hac10**] c_1 is gained. Let $H_0^1(\Omega) \subset V \subset H^1(\Omega)$ and $a(.,.)$ a V-elliptic bilinear-form. The variational formulation for $u \in H_1$

$$a(u, v) = (f, v)_0, \qquad \forall v \in V$$

is called H^2-regular if there exists for all $f \in H^0(\Omega)$ a solution $u \in H^2(\Omega)$ and additionally there is a constant $c_1(\Omega, a, s)$ fulfilling

$$\|u\|_2 \leq c_1 \|f\|_0. \tag{3.20}$$

s denotes the Poincaré constant that fulfills the Poincaré-Friedrichsche inequality

$$\|v\|_0 \leq |v|_1, \qquad \forall v \in H_0^1(\Omega)$$

where Ω is in-closed by a cube of edge length s. There are several other estimates for the Poincaré constant according to [**GalXX**], which are e.g. based on circles:

$$s \leq \frac{d}{\sqrt{2}\pi}.$$

Here, the domain Ω can be surrounded by a circle with diameter d.

Finally the constant c_2 arises in the context of the quality of the interpolation. A quasi-uniform triangulation τ_h of Ω is assumed. For an interpolation based on piecewise polynomials of degree $t - 1$, with $t > 1$, and a constant $c_2(\Omega, s, t)$ holds

$$\|u - I_h u\|_{2,h} \leq c_2 \cdot h^{t-2}|u|_{t,\Omega} \qquad \forall u \in H^t(\Omega). \tag{3.21}$$

The elements have to contain a circle with radius $\rho_\tau \geq \frac{h}{s}$ [**Bra07**].

3.3.4. Determining the Constants - an Example. For determining the constants that influence the choice of a floating point format, the estimation (3.19) is considered and the constants are approximated for a simple example where the following Poisson equation is considered:

$$\left.\begin{aligned} -\Delta u &= f \ \text{ in } \Omega \\ u &= 0 \text{ on } \partial\Omega \end{aligned}\right\}$$

where

$$\begin{aligned} f &= 2 \cdot k^2\pi^2 \sin(k\pi x)\sin(k\pi y), \\ \Omega &= [0,k]^2 \subset \mathbb{R}^2. \end{aligned}$$

This problem has the analytically exact solution $u(x,y) = \sin(k\pi x)\sin(k\pi y)$.

For the discretization step, an uniform decomposition of the domain Ω into $2\left(\frac{k}{h}\right)^2$ triangles is used.

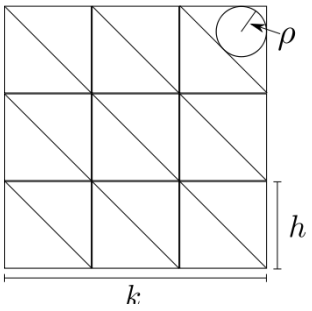


Figure 3.8. Decomposition of Ω.

The domain Ω can be enclosed by a circumcircle with diameter $d = k\sqrt{2}$, the triangles have an incircle with radius $\rho = \frac{h}{2+\sqrt{2}}$.

For the further estimations, c_s and c_a need to be approximated. Using the definition of the continuity of $a(\cdot,\cdot)$ (see 3.13), it is possible to estimate

$$0 \leq \underbrace{\frac{|a(u,u)|}{\|u\|_1^2}}_{*} \leq C$$

based on the availability of the solution u. It becomes visible that $*$ can be interpreted as some kind of Rayleigh quotient and by using the solution u one can compute a lower bound for the continuity constant:

$$|a(u,u)| = \int_0^k \int_0^k \nabla u \nabla u dxdy = \frac{1}{2}k^2\pi^2 - \frac{1}{2}cos^2(k^2\pi)sin^2(k^2\pi) \tag{3.22}$$

$$\begin{aligned} \|u\|_1^2 &= \int_\Omega u + \int_\Omega \nabla u \nabla u dxdy \qquad (3.23) \\ &= \underbrace{\underbrace{\int_0^k \int_0^k sin(k\pi x)sin(k\pi y)dxdy}_{\frac{1-cos(k^2\pi)+cos^2(k^2\pi)}{\pi^2k^2}} + \underbrace{\frac{1}{2}k^4\pi^2 - \frac{1}{2}cos^2(k^2\pi)sin^2(k^2\pi)}_{see 3.23}}_{***} \end{aligned}$$

So one gets as estimation for the lower bound of the continuity constant:

$$\frac{|a(u,u)|}{\|u\|_1^2} = \frac{**}{***} \overset{k=1}{=} \frac{\pi^4}{\pi^4+8} \approx 0,92.$$

In case of c_a, information about the coercitivity constant α is needed and the Poincaré constant s has to be approximated. This can be achieved by using the circumcircle radius of the domain Ω, and the relation $s \leq \frac{d}{\sqrt{n}\pi}$, where n is the space dimension [**GalXX**]. Using $s \leq \frac{k}{\pi}$ it is possible to derive

$$\begin{aligned}
\|v\|_1^2 &= (\|v\|_0 + \|\nabla v\|_0)^2 \\
&\leq (1+s)^2 \|\nabla v\|_0^2 \\
&= (1+s)^2 \underbrace{\int_\Omega \nabla v \cdot \nabla v \; dx \; dy}_{=a(v,v)} \\
\Rightarrow \alpha &\leq \frac{1}{(1+s)^2}.
\end{aligned}$$

The H_0 norm of the right-hand side is also needed:

$$\begin{aligned}
\| \; f \; \|_0 \; &= \; \sqrt{\int_0^k \int_0^k f \cdot f \; dx \; dy} \\
&= \; \sqrt{\int_0^k \int_0^k (2k^2\pi^2 \sin(k\pi x)\sin(k\pi y))^2 \; dx \; dy} \\
&= \; \sqrt{2 - 4cos(k^2\pi) + 2(cos(k^2\pi))^2} \\
&\overset{k=1}{=} 2\sqrt{2}.
\end{aligned} \tag{3.24}$$

Finally, one is left with the constants c_1 and c_2, that are usually difficult to approximate. For the further approximations, $k = 1$ is chosen, and one obtains for the domain Ω the unit square.

In case of c_1, the definition of the H^2-regularity is used (see 3.20) to approximate

$$c_1 \approx 0.051.$$

For the constant c_2 (see 3.21) it is assumed to have the interpolation function $Iu \equiv 0$ and the coarse approximation for c_2 is (construction-conditioned analogue to c_1)

$$c_2 \approx 0.051.$$

By combining all information from above and by assuming that the approximation of the bilinearform in the high floating point format is equal to the analytical one, meaning $a(\cdot,\cdot) = \tilde{a}(\cdot,\cdot)$, one can calculate from equation 3.19 the perturbation that is allowed in order to keep the method error smaller then the discretization error (for the chosen example and certain assumptions):

$$\sup_{w_h \in V_h} \left(\frac{|b_1|}{\|w_h\|_m} \right) + \sup_{w_h \in V_h} \left(\frac{|b_2|}{\|w_h\|_m} \right) \leq \frac{c_a}{c_s} h^2 \|f\| = \frac{h^2(1+c_1)c_2 C \|f\|_0}{(1+C)\alpha} \overset{k=1}{\approx} 0,068h^2.$$

From this equation it becomes also visible that the finer the discretization width h is chosen, the perturbation decreases by h^2, indicating that the condition of the problem becomes worse.

3.3.5. Conclusion. Within this section, the numerical analysis for the perturbation that can be applied to discretizations of elliptic operators, discretized with finite element methods, is explained. The error estimation, based on a lemma of Strang and the lemma of Aubin-Nitsche, gives strong upper bounds for the perturbation in order to keep the method error (at most) in the order of magnitude of the discretization error. Since the constants depend on the problem that is solved, approximations can usually only be obtained through numerical experiments. In

combination with the theory around a posteriori error estimators, this theory offers multiple application possibilities.

Multiple perspectives arise in the context of the presented findings. Solvers could be developed which iteratively approximate the bounds for the allowed perturbation and variegate the employed floating point formats according to the bounds. Hardware architectures allowing such techniques without using abstraction layers are FPGAs. Software sided it is possible to use libraries offering the possibility to define arbitrary precision formats like the GMP library [**Lib**].

CHAPTER 4

Implementations of CG and GMRES on Dedicated Hardware

This chapter evaluates the properties of the three in Chapter 2.2 explained paradigms MPI, OpenMP and CUDA in the context of the two solvers CG (see [**Saa03**] or Chapter 1.3) and GMRES (see [**Saa03**] or Chapter 1.4) based on experiments with eight matrices. Most of them are from 2D and 3D fluid flow problems that have been discretized by finite elements. They are stored in the CSR-format on which the implementations work. Considered are implementations which are based on a single paradigm, as well as hybrid ones, often denoted as hybrid strategies. Additionally, an implicit parallelization using a highly optimized library offering basic linear algebra subroutines (BLAS) is taken into account. Preconditioning, which is extremely depending on the problem, is shown in Section 4.9. As a reference with respect to performance, PETSC [**BBG$^+$09, BBE$^+$08, BGMS97**] has been chosen.

Since the development of modern technology is characterized by simulations, that are often no longer performed through physical experiments, but through mathematical modeling and numerical simulation. For many simulations, for example in computational fluid dynamics, enormous computation power and memory is needed to be able to handle the often arising very large systems of linear equations.

Parallelization is nowadays the solution for coping with such tasks, as shown in the introduction. There are many paradigms for programming parallel CPU-based architectures, where the most common ones are MPI and OpenMP. For accessing the computational power of a GPU, CUDA is widely used. Every paradigm has its own advantages and disadvantages and sometimes it is not easy for software developers to choose the optimal combination for the implementations. Results from this Chapter can also be found in [**Gal10**] and are submitted for publication.

4.1. Data Storage Formats

This section explains a selection of storage formats for matrix data. Beside the Coordinate Formate, the Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) format are illustrated and an example is shown.

Storing only the non-zero entries of a matrix is the basic idea of sparse data formats in order to save memory space when only a minor number of the entries in a matrix differ from zero. In general, the more memory for storage is used, the more structured are the access patterns to the matrix entries. Therefore one should find a trade-off between memory usage and element access structure.

The used format is described based on the following example:

$$A = \begin{bmatrix} 1 & -1 & 0 & 0 & 3 \\ 0 & 2 & 2 & 0 & 0 \\ 0 & 11 & 3 & -4 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 3 & 1 & 0 & 5 \end{bmatrix}. \tag{4.1}$$

The CSR- format is also known as Compressed Row Storage (CRS). The idea is to store only non-zero entries of a matrix. Therefore the position of the non-zero entry has to be saved too. A matrix is described by the three arrays *val*, *col* and *row*. All non-zero entries are stored in *val* and its column numbers are stored in *col* and array *row* contains pointers to the first elements in every row. Usually *val* is of type *double* or *float* with length nnz (number of non-zeros), *col* and *row* are of type *integer* with length nnz respectively $\dim(A) + 1$. In case of the example 4.1, the arrays are as shown in Figure 4.1. The indexing is zero-based (like in C/C++

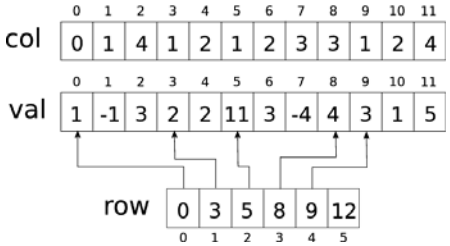


Figure 4.1. CSR - format for matrix 4.1

etc.), but one-based indexing (like in Fortran etc.) is also possible. For symmetric matrices the memory usage can be decreased by storing only the upper or lower triangular matrix.

Beside the CSR-format, there exist lots of storage formats, each with application- and hardware-specific advantages and drawbacks.

4.2. Parallel CG Implementations

4.2.1. OpenMP-Parallelization. Considered are two OpenMP implementations of CG. The first implementation extends the sequential algorithm with loop parallelization by OpenMP with an implicit data distribution. The second implementation of CG is based on a (logical) data distribution. For this reason the matrix and all vectors are divided into blocks. All matrix blocks and vectors are distributed uniformly over the threads but there is no physical data distribution. All threads have access to the same data except for such that is declared as private. Communications occur via memory respectively shared variables. Finally, These two implementations are compared with a sequential implementation utilizing the BLAS-implementations from the Intel MKL which use thread parallelization and are highly optimized for the underlying hardware from the same vendor.

It is possible to parallelize the matrix-vector product, dot products and vector updates. For the computation of α and β a single region is needed to avoid that all threads compute the same value and reduce the computation overhead. Additionally, on more modern hardware architectures (like Intel Nehalem etc.), the clock frequency of one core raises if the other cores are not used in order to speedup sequential operations.

1.	$v_i = Ap_i$	*parallel*
2.	$sp\_pv = p_i^T v_i$	*parallel*
3.	$\alpha_i = \frac{sp\_r}{sp\_pv}$	*sequential*
4.	$r_{i+1} = r_i + \alpha_i v_i$	*parallel*
5.	$sp\_r = r_{i+1}^T r_{i+1}$	*parallel*
6.	$\beta_i = \frac{sp\_r}{sp\_r\_old}$	*sequential*
7.	$sp\_r\_old = sp\_r$	*sequential*
8.	$x_{i+1} = x_i + \alpha_i p_i$	*parallel*
9.	$p_{i+1} = -r_{i+1} + \beta_i p_i$	*parallel*

Table 4.1. OpenMP parallelizing scheme for a CG iteration cycle

4.2.2. MPI-Parallelization. First a distribution strategy for the matrix and vectors has to be chosen. The used approach is to distribute the matrix A and the vectors by rows (see figure 4.2), which is the most intuitive distribution when CSR is the data storage format. The number of rows of the vectors are equal to the number of rows on every process. Therefore the vector p has to be communicated among all processes to perform the local matrix-vector products. The vectors x, r and v need not to be communicated. Furthermore the dot products $r^T r$ and $p^T Ap$ have to be communicated to compute α and β. This data distribution is similar to a two dimensional block-cyclic data distribution based on a $n \times 1$ grid of processes which is analyzed in Chapter 6 in terms of performance and accuracy. Since the focus in this chapter is performance, the fastest data distribution (on the used hardware) has been chosen.

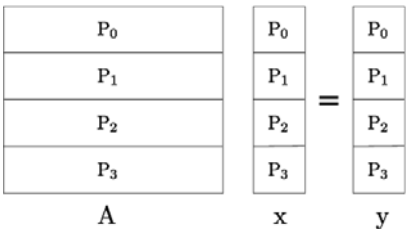


Figure 4.2. MPI matrix distribution.

There are different designs for implementing the CG algorithm. In the shown implementation, all processes perform the same operations on different data (SPMD). The vector p is communicated with two different strategies. One is a so called neighbor communication, where a process k communicates only with the processes $k-1$ and $k+1$. Note that this is only possible when the data needed by k is available from the direct neighbors. One example for such a case is when the bandwidth of the data a process holds is smaller then the number of rows (using CSR) the neighbor process holds. To avoid the overhead of unnecessary synchronization, asynchronous communication is used. Finally, the dot products are communicated using the MPI routine *MPI_Allreduce()*. Table 4.2 shows the algorithm design of the implementation. Furthermore, for the sequential runtime the best sequential result of the compared implementations was chosen to calculate the speedup. As reference implementation in terms of performance, the PETSc CG solver was used to benchmark the presented parallelization strategies.

4.2.3. Hybrid-Parallelization. Both approaches from the two previous sections are combined to obtain a hybrid implementation. The MPI implementation above is extended with an OpenMP loop parallelization for the local matrix-vector

1.	communicate/compute p	
2.	$v_i = Ap_i$	*parallel*
3.	$sp\_pv = p_i^T v_i$	*parallel*
4.	communicate sp_pv	
5.	$\alpha_i = \frac{sp\_r}{sp\_pv}$	*local*
6.	$r_{i+1} = r_i + \alpha_i v_i$	*parallel*
7.	$sp\_r = r_{i+1}^T r_{i+1}$	*parallel*
8.	communicate sp_r	
9.	$\beta_i = \frac{sp\_r}{sp\_r\_old}$	*local*
10.	$sp\_r\_old = sp\_r$	*local*
11.	$x_{i+1} = x_i - \alpha_i p_i$	*parallel*
12.	$p_{i+1} = r_{i+1} + \beta_i p_i$	*parallel*

Table 4.2. MPI parallelizing scheme for a CG iteration cycle

multiplication, the local vector updates and the local dot products (see table 4.3). Alternatively it is possible to use the multithreaded MKL to implement an hybrid version of CG.

1.	$v_i = Ap_i$	*parallel/hybrid*
2.	$sp\_pv = p_i^T v_i$	*parallel/hybrid*
3.	communicate sp_pv	
4.	$\alpha_i = \frac{sp\_r}{sp\_pv}$	*local*
5.	$r_{i+1} = r_i + \alpha_i v_i$	*parallel/hybrid*
6.	$sp\_r = r_{i+1}^T r_{i+1}$	*parallel/hybrid*
7.	communicate sp_r	
8.	$\beta_i = \frac{sp\_r}{sp\_r\_old}$	*local*
9.	$sp\_r\_old = sp\_r$	*local*
10.	$x_{i+1} = x_i + \alpha_i p_i$	*parallel/hybrid*
11.	$p_{i+1} = -r_{i+1} + \beta_i p_i$	*parallel/hybrid*
12.	communicate p	

Table 4.3. Hybrid parallelizing scheme for a CG iteration cycle

4.2.4. CUDA-Parallelization. Finally, a CG implementation on a NVIDIA GPU using CUDA is described. Implement is the same algorithm design as for the CPU versions using CUBLAS and an extra kernel for the sparse matrix-vector multiplication. The CPU passes the data via PCIs to the GPU, where all computations are done. All CUDA-implementation were executed on a Tesla S1070 based system which is a representative from the professional GPU-computing series of NVIDIA .

4.3. Parallel GMRES implementations

In this section a strategy for GMRES(m) parallelization in different environments is presented. As explained in Chapter 1.4, the implementation of pure GMRES is usually not practicable. Therefore the focus is set on a restarted GMRES(m) and its parallelization. See e.g. [**SS86**] or Chapter 1.4 for detailed information. As restart parameter, $m = 20$ was set for all experiments.

4.3.1. OpenMP-Parallelization. Two pure OpenMP implementations and a MKL implementation using the MKL thread parallelism are evaluated. The parallelization strategy for the pure OpenMP approaches are the same as for CG. First

loops within the algorithm are parallelized and in the second implementation a logical data distribution is used.

4.3.2. MPI-Parallelization. The same data distribution as in CG parallelization is chosen, that is a distribution by blocks of rows. Chosen is the strategy to parallelize the nested Arnoldi algorithm and the solution update at the end of GMRES(m), while the least-squares problem is not solved in parallel. Furthermore this least-squares problem is solved by every process which is motivated by the fact that the restart parameter m is usually very small (5-50). For this reason it creates to much overhead to distribute this small problem over all processes.

1.	Arnoldi algorithm	*parallel*
	a. sparse matrix vector product	*parallel with comm.*
	b. dot products	*parallel with comm.*
	c. vector updates	*parallel*
2.	solving the least squares problem	*sequential*
3.	solution update	*parallel*

Table 4.4. MPI parallelization scheme for a GMRES iteration cycle

The parallel Arnoldi algorithm consists of parallel sparse matrix-vector multiplications, parallel vector updates and parallel dot products. To perform such operations, parallel MPI-based implementations are used.

4.3.3. Hybrid-Parallelization. In the proposed hybrid implementation MPI and OpenMP are mixed. The plain MPI solver is extended with a local OpenMP parallelization on every process and a hierarchical parallelization can be obtained.

1.	Arnoldi algorithm	*parallel/hybrid*
	a. sparse matrix vector product	*parallel/hybrid with comm.*
	b. dot products	*parallel/hybrid with comm.*
	c. vector updates	*parallel*
2.	solving the least squares problem	*sequential*
3.	solution update	*parallel/hybrid*

Table 4.5. Hybrid parallelization scheme for a GMRES iteration cycle

4.3.4. CUDA-Parallelization. A significant difference can be seen by comparing the CUDA implementation approach for GMRES to the approach for CG. While the CG solver runs completely on the GPU, the GMRES solver uses the GPU as an accelerator to speedup some of the performed operations. In fact, the least-squares problem within the restarted GMRES algorithm is solved on the CPU, because the system is to small to use the GPU efficiently. The dot products, vector updates are performed by using CUBLAS routines on the GPU. The sparse matrix-vector multiplication is also done on the GPU by the use of a special kernel (meanwhile CUBLAS offers a sparse matrix-vector multiplication, which was not the case when the experiments were done). Therefore the dot product results within the Arnoldi algorithm must be send via PCIe to the host. After adjusting and solving the least-squares problem the least-squares solution is given back to the device for updating the approximate solution.

4.4. Reference Examples

As a first test-case a finite difference discretization of the Laplacian equation in 2D on the unit square is chosen that is based on a 5-point stencil. The next three test-cases arise in the context of a finite element discretization of a Venturi nozzle in two and three dimensions with Q2-elements for the velocities and Q1-elements for the pressure. Additionally the test-matrices CFD ACUSIM, Thermal 2, Stomach and CFD Rothberg are taken from the University of Florida Sparse Matrix Collection [**Dav**]. Estimations for the condition number have been performed by using the Matlab function "condest()" which gives back a 1-norm estimate. As right hand side for the experiments, the product of the test-case with the vector $x := (1, \ldots, 1)^T$ was chosen. An absolute stopping criteria of $\|r_k\| \leq 10^{-10}$ for the residual was chosen to stop the solver except for the test-cases Venturi 2D 3 and CFD Venturi 3D, where $\|r_k\| \leq 10^{-6}$ was taken.

Laplace 2D
$A = \begin{pmatrix} B & -I & & \\ -I & B & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & B \end{pmatrix} \in \mathbb{R}^{mn \times mn}$, with $B = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{n \times n}$.
Problem: Laplace 2D, FDM Problem size: $n = n * m$ Sparsity: $nnz = m(3n-2) + 2n(m-1)$ Symmetric: yes Pos. define: yes

Table 4.6. Sparsity plot and property of the test-case Laplace 2D.

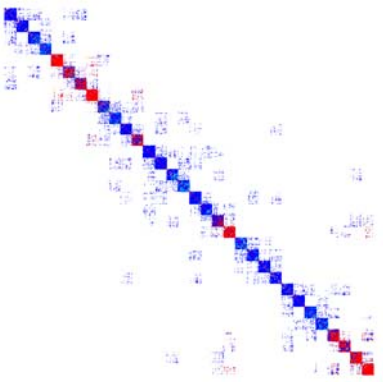
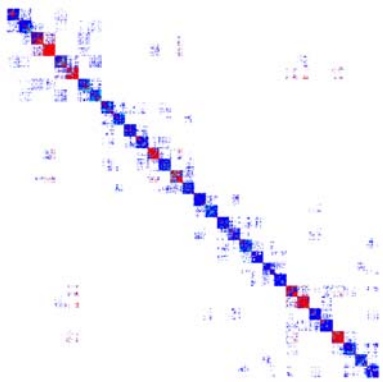
CFD Venturi 2D 1	**CFD Venturi 2D 3**
	
Problem: CFD Venturi 2D, FEM Problem size: $n = 395.009$ Sparsity: $nnz = 3.544.321$ Symmetric: no Pos. define: ?	Problem: CFD Venturi 2D, FEM Problem size: $n = 1.019.967$ Sparsity: $nnz = 9.182.401$ Symmetric: no Pos. define: ?

Table 4.7. Sparsity plots and properties of the matrices CFD Venturi 2D.

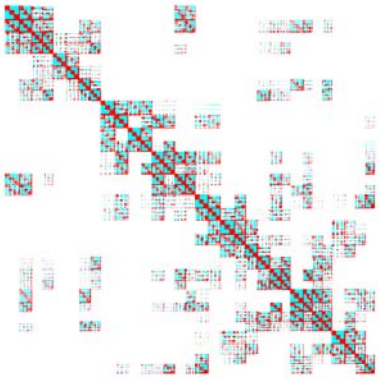
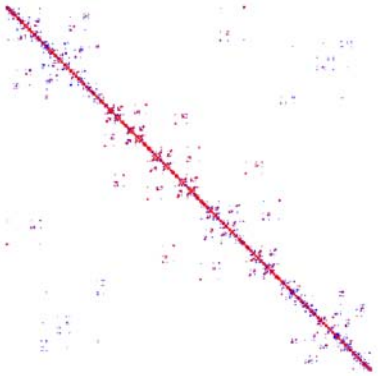
CFD Venturi 3D	CFD ACUSIM
	
Problem: CFD Venturi 3D, FEM Problem size: $n = 106.496$ Sparsity: $nnz = 21.510.676$ Symmetric: no Pos. define: ?	Problem: CFD ACUSIM Problem size: $n = 14.822$ Sparsity: $nnz = 365.313$ Symmetric: yes Pos. define: yes Estimated cond.: $3,2 \cdot 10^6$

Table 4.8. Sparsity plots and properties of the test-matrices CFD Venturi 3D and ACUSIM.

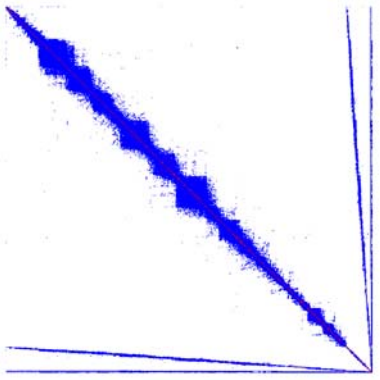
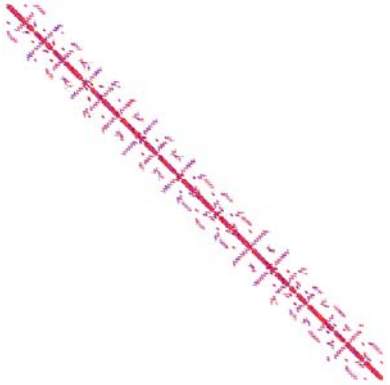
Thermal 2	Stomach
	
Problem: Thermal problem, FEM Problem size: $n = 1.228.045$ Sparsity: $nnz = 8.580.313$ Symmetric: yes Pos. define: yes Estimated cond.: $7,5 \cdot 10^6$	Problem: Human Duodenum 3D Problem size: $n = 213.360$ Sparsity: $nnz = 3.021.648$ Symmetric: no Pos. define: no Estimated cond.: $85,2$

Table 4.9. Sparsity plots and properties of the matrices Thermal 2 and Stomach.

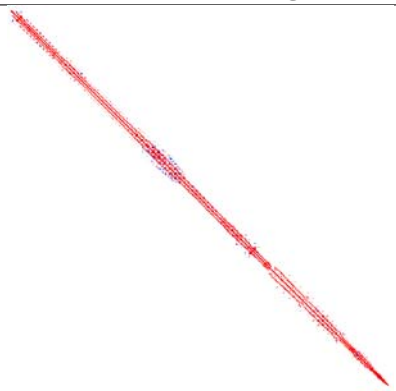
CFD Rothberg

Problem: Computational fluid dynamics Problem size: $n = 123.440$ Sparsity: $nnz = 3.085.406$ Symmetric: yes Positive define: yes Estimated cond.: $3,7 \cdot 10^6$

Table 4.10. Sparsity plot and properties of the test-matrix Rothberg.

4.5. Hard- and Software Environment

Two hardware platforms are used for the experiments. One is a TESLA based system equipped with one NVIDIA TESLA S1070. Each of the two host nodes is connected via a PCIe 2.0 x16 to the S1070 and are each equipped with two Intel Quad-core Xeon 5450 CPUs running at 3.0 GHz. The software platform adopted for the numerical experiments is composed of the Intel MKL in version 10.1.1.019 and the Intel compiler in version 11.0.074.

The second platform is the Institutscluster which is located at the Steinbuch Centre for Computing (SCC) at the Karlsruhe Institute of Technology (KIT). It consists of 200 computing nodes each equipped with two Intel quad-core EM64T Xeon 5355 processors running at 2,6 GHZ, 16 GB of main memory and an Infiniband 4x DDR interconnect. The overall peak performance of the whole system is about 17,57 TFlops and 15,2 TFlops in the Linpack benchmark. In addition to the Intel MKL in version 10.1.2.024, the Intel compiler in version 11.1.056 and OpenMPI 1.3.1 are chosen.

4.6. Numerical Experiments based on CG

4.6.1. OpenMP-Parallelization. The OpenMP implementation for the Laplace problem with 90.000 unknowns scales well, since the problem fits complete into the cache. In Figure 4.4 a trip between four and five threads can be observed, which is caused by the architecture of the Institutscluster where the eight available cores are divided on two separate sockets. In the situation of four threads the program runs on a single quad-core processor, while in the case of five threads the program runs on both quad-core processors. A manual pinning of threads to dedicated cores is possible but was not taken into account within this work. The distribution of threads was left to the operating system. In all test cases, except Laplace with 90.000 unknowns, a performance limitation by the memory bandwidth on a IC1 node can be

observed. Furthermore, there is no difference in the performance between the loop parallelized and the data distributed implementation.

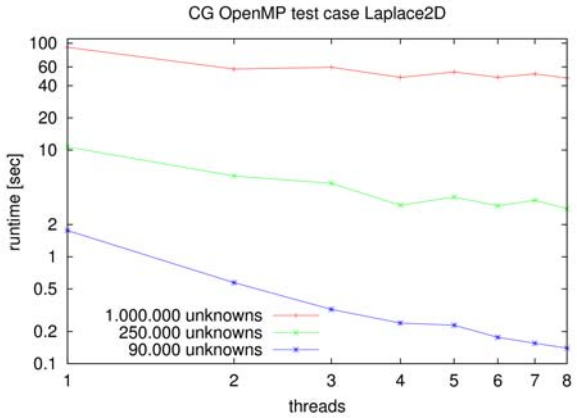


Figure 4.3. Test case Laplace 2D: Results for the loop parallelized CG implementation with different problem sizes.

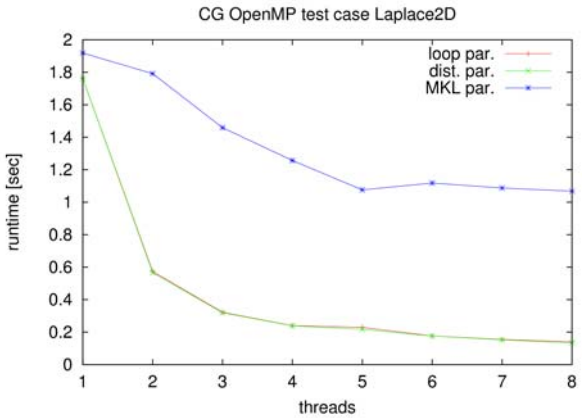


Figure 4.4. Test case Laplace 2D with 90.000 unknowns: Comparison of the loop parallelized, data distributed and MKL threaded OpenMP implementations.

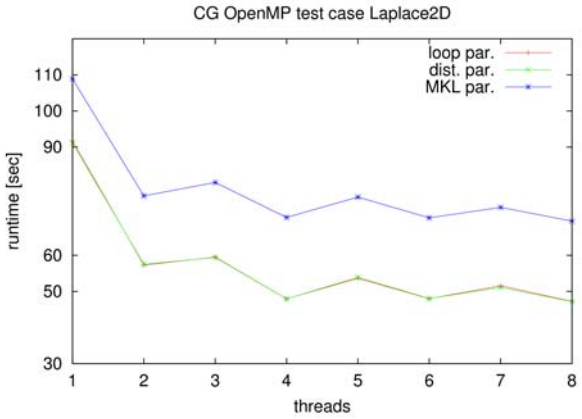


Figure 4.5. Test case Laplace 2D with 1.000.000 unknowns: Comparison of the loop parallelized, data distributed and MKL threaded OpenMP implementations.

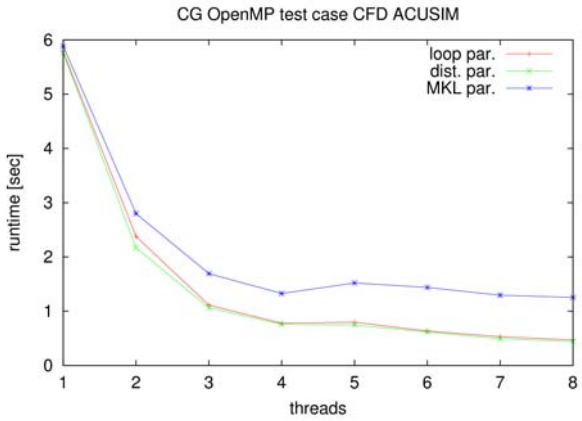


Figure 4.6. Test case CFD ACUSIM: Comparison of the loop parallelized, data distributed and MKL threaded OpenMP implementations.

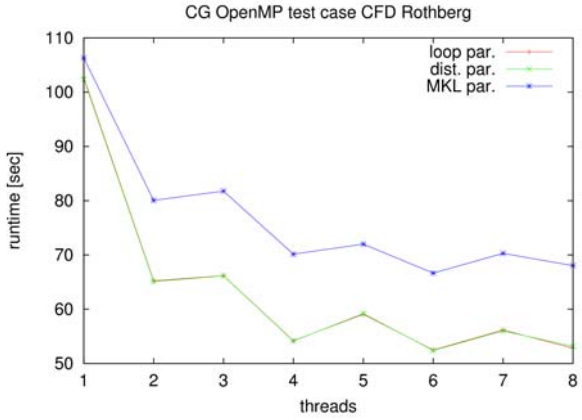


Figure 4.7. Test-case CFD Rothberg: Comparison of the loop parallelized, data distributed and MKL threaded OpenMP implementations.

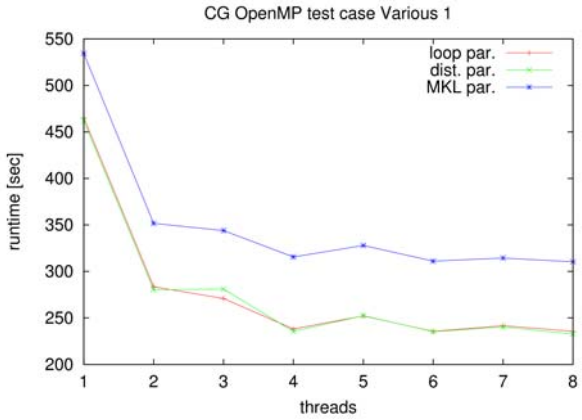


Figure 4.8. Test-case Thermal 2: Comparison of the loop parallelized, data distributed and MKL threaded OpenMP implementations.

4.6.2. MPI-Parallelization. As reference in terms of performance, the results of the above described implementations are compared to ones of the PETSc

library. How rounding errors in combination with parallelization can effect accuracy of elementary operations is explained in Chapter 6 and effects on solvers can be found in Chapter 7. The MPI-based CG solver scales well and for the Laplace test-case the performance is excellent (Figures 4.9-4.14).

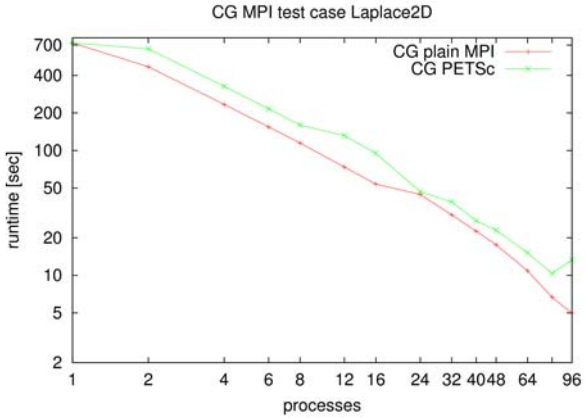


Figure 4.9. Test case Laplace2D with 4.000.000 unknowns. Comparison of two plain MPI Implementations of CG with PETSc. Two processes per node and neighbour communication.

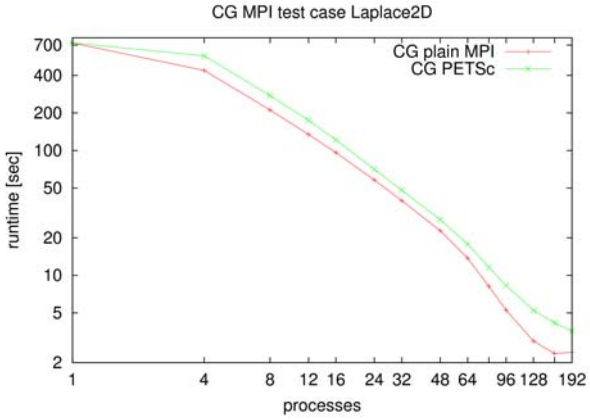


Figure 4.10. Test case Laplace2D with 4.000.000 unknowns. Comparison of two plain MPI Implementations of CG with PETSc. Four processes per node and neighbour communication.

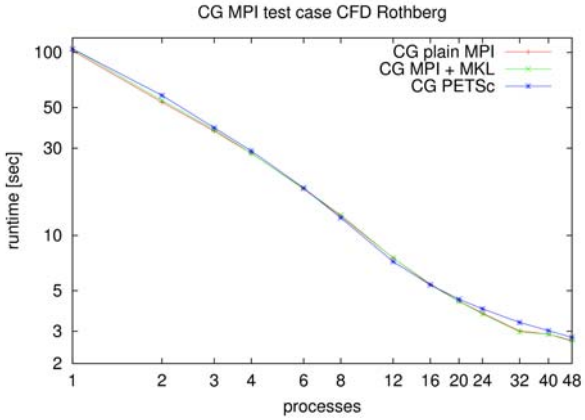


Figure 4.11. Test-case CFD Rothberg with one single-threaded process per node. Selected gather is used.

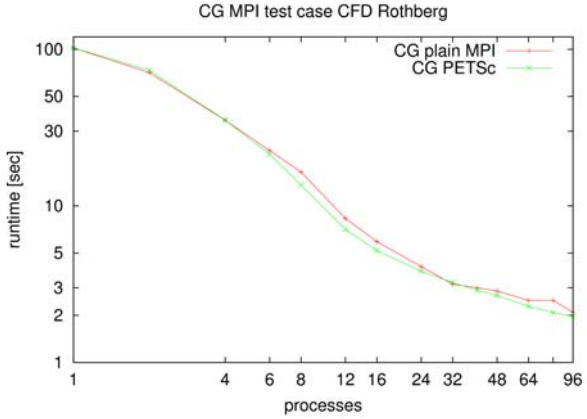


Figure 4.12. Benchmark for the test-case CFD Rothberg with two single-threaded processes per node. Selected gather is used.

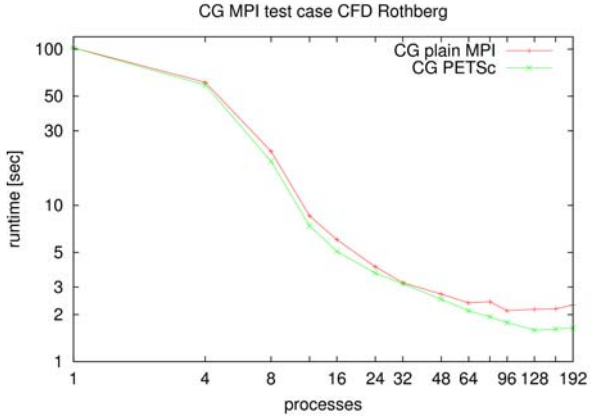


Figure 4.13. Benchmark for the test case CFD Rothberg with four single-threaded processes per node. Selected gather is used.

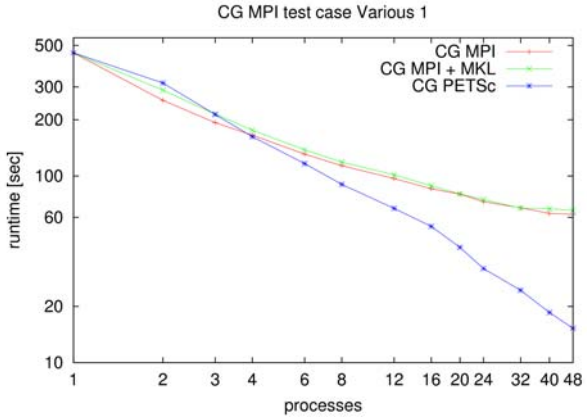


Figure 4.14. Results for the test case Thermal 2. Selected gather is used.

The CG solver shows a super linear speedup for more than 24 processes on a single node for the Laplace 2D test case, because then the problem fits into the caches.

4.6.3. Hybrid-Parallelization. The results seen in figures 4.17, 4.18, 4.20 and 4.21 show that a mixed OpenMP/MPI implementation has no perfomance benefit in the test environment compared to a pure MPI implementation.

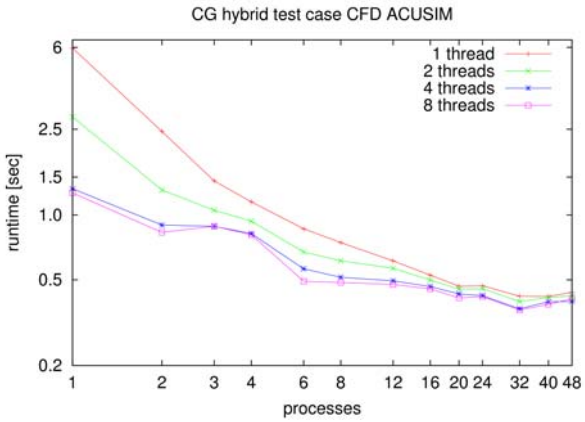


Figure 4.15. Test case CFD ACUSIM with MKL multithreaded processes. Each processes ran exclusively on a single node.

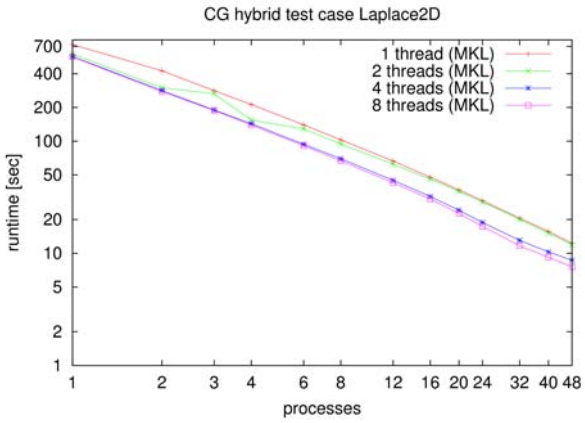


Figure 4.16. Test case Laplace2D with MKL multithreaded processes. Each processes ran exclusively on a single node.

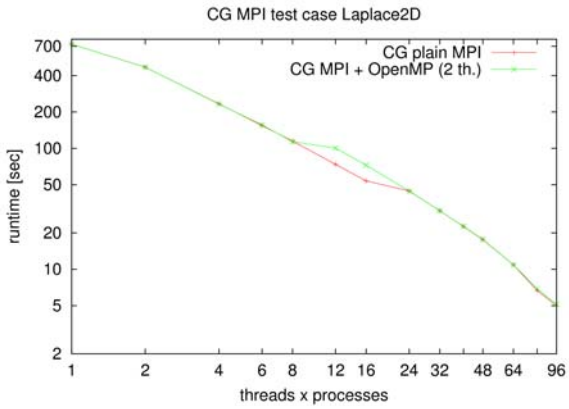


Figure 4.17. Comparison of a plain MPI and a hybrid solver for the test case Laplace2D. The hybrid implementation uses two threads in one process on one node and the plain MPI implementation uses two single-threaded processes on one node.

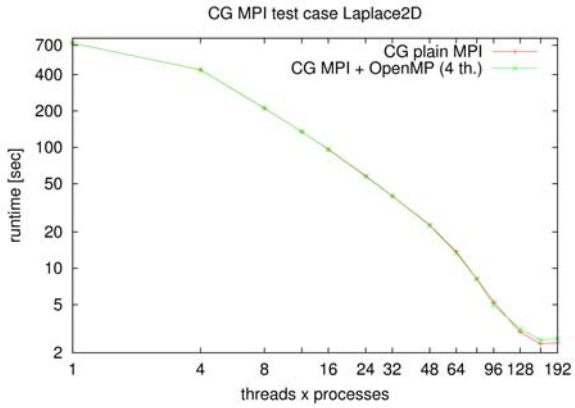


Figure 4.18. Comparison of a plain MPI solver and a hybrid solver for the test case Laplace2D. The hybrid implementation uses four threads in one process on one node, the plain MPI implementation uses four single-threaded processes on one node.

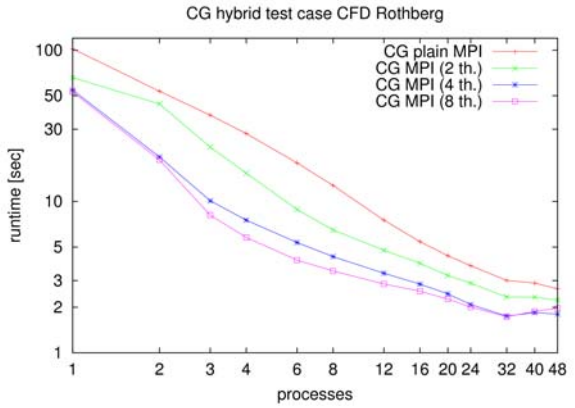


Figure 4.19. Test case CFD Rothberg with plain OpenMP multithreaded processes. All processes ran on a single node.

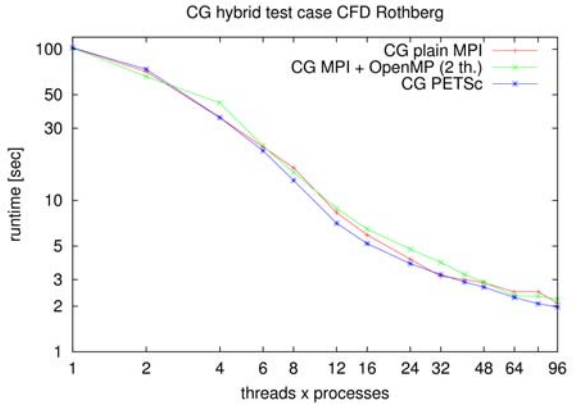


Figure 4.20. Comparison of a plain MPI solver and a hybrid solver for the test case CFD Rothberg. The hybrid implementation uses two threads in one process on one node and the plain MPI implementation uses two single-threaded processes on one node.

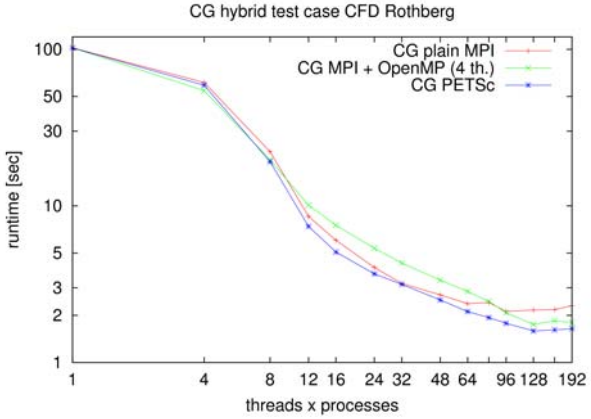


Figure 4.21. Comparison of a plain MPI solver and a hybrid solver for the test case CFD Rothberg. The hybrid implementation uses four threads in one process on one node and the plain MPI implementation uses four single-threaded processes on one node.

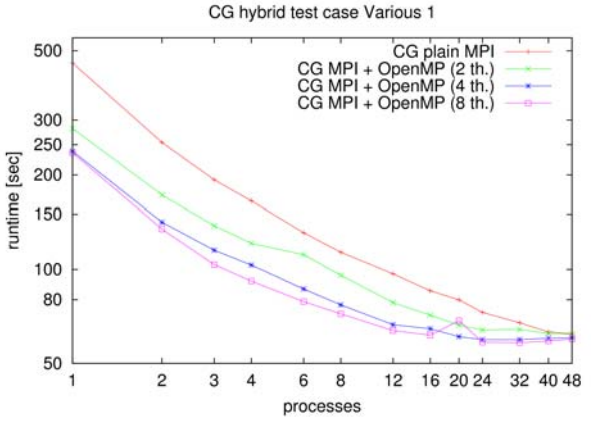


Figure 4.22. Test case Thermal 2 with plain OpenMP multi-threaded processes. All processes ran on a single node.

4.6.4. CUDA-Parallelization. Results for the CUDA implementation are in shown in Table 4.11. The speedup is with respect to the sequential results on the CPU for the related test-case. In Figure 4.24 the performance of the GPU

Test case	Runtime [sec]	Speedup
Laplace 2D (4.000.000 unknowns)	108,63	7,80
Laplace 2D (1.000.000 unknowns)	14,26	6,38
Laplace 2D (250.000 unknowns)	2,06	5,17
CFD ACUSIM	2,71	2,20
CFD Rothberg	35,37	2,96
Thermal 2	85,98	6,16

Table 4.11. Benchmark results for CUDA CG.

solver is compared to a pure MPI implementation to classify the performance of the CUDA implementation. The MPI solver with eight processes (running on four

nodes) achieves the performance of the GPU solver, whereas the amount of employed ressoures (in terms of energy and costs) is significantly higher.

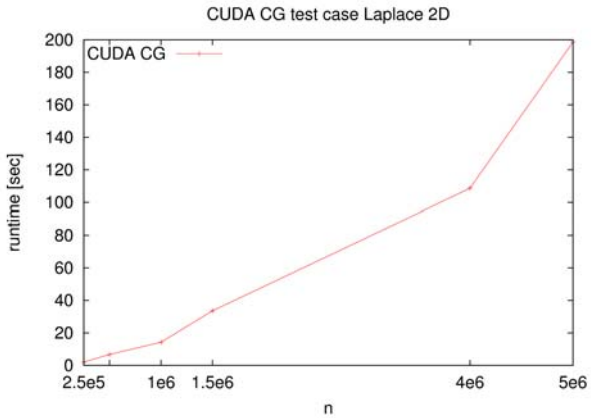


Figure 4.23. CUDA CG for the test case Laplace 2D with different number of unknowns.

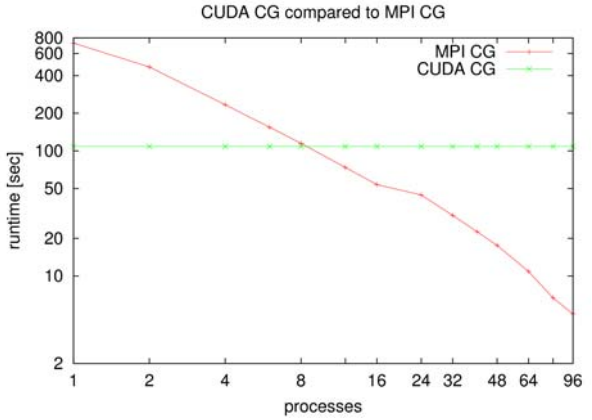


Figure 4.24. CUDA CG compared with a MPI implementation of CG, where two processes ran on one IC1 node. Test-case is the Laplace 2D matrix with four million unknowns.

4.7. Numerical Experiments Based on GMRES

4.7.1. OpenMP-Parallelization. The three GMRES(m) implementations show the same scaling performance problems as the CG implementations. This is also due to the limited memory bandwidth on one Institutscluster node.

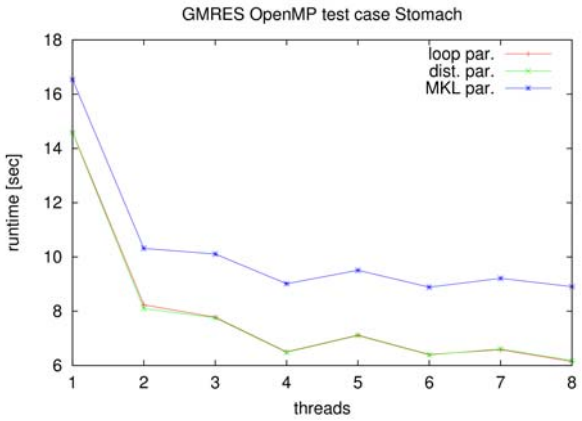


Figure 4.25. Results for the test case Stomach. Three implementations are compared: OpenMP loop parallelization, OpenMP distributed data parallelization and MKL thread parallelization.

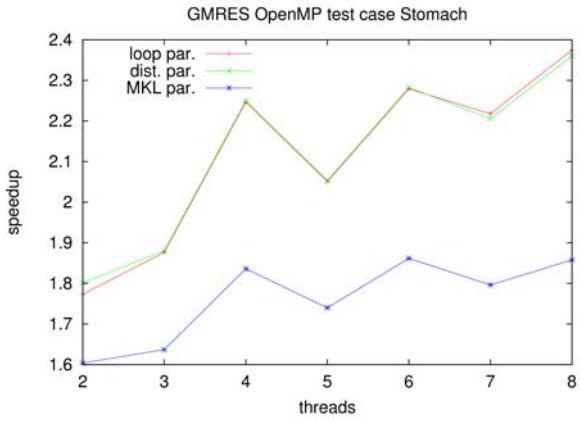


Figure 4.26. Speedups for the test case Stomach.

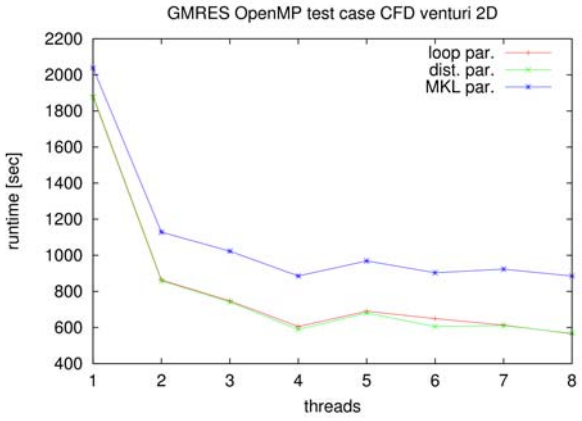


Figure 4.27. Results for the test case CFD Venturi 2D 1. Three implementations are compared: OpenMP loop parallelization, OpenMP distributed data parallelization and MKL thread parallelization.

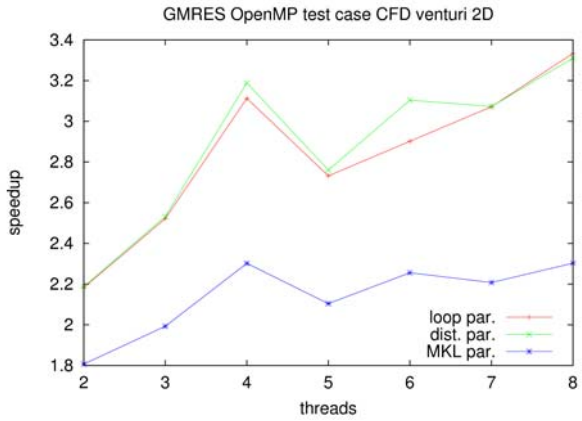


Figure 4.28. Speedups for the test case CFD Venturi 2D 1.

4.7.2. MPI-Parallelization. Compared to the PETSc solver the proposed solver has massive scaling problems. There is a better communication implementation needed.

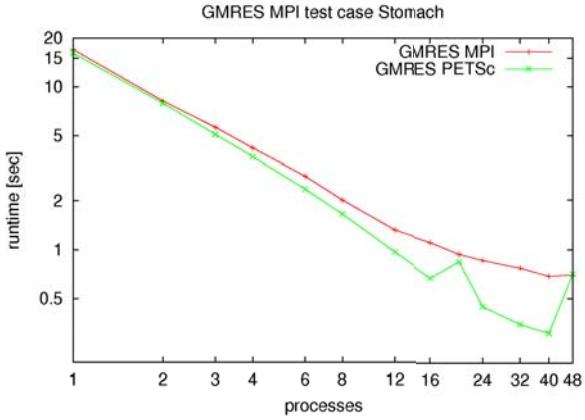


Figure 4.29. Results for the test case Stomach. Selected gather is used.

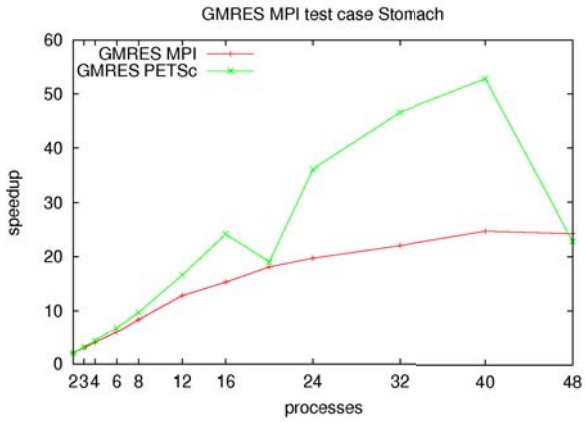


Figure 4.30. Speedups for the test case Stomach. Selected gather is used.

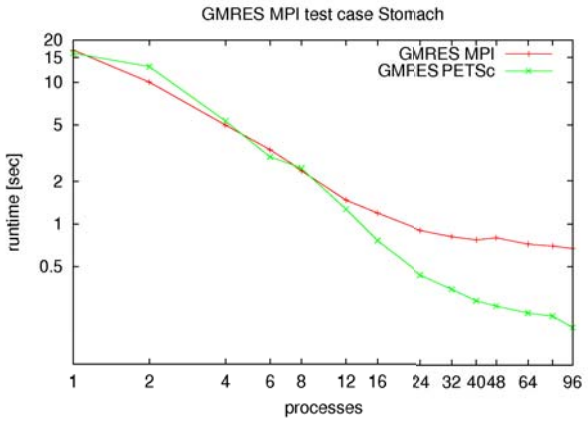


Figure 4.31. Results for the test case Stomach with two processes per IC1 node. Selected gather is used.

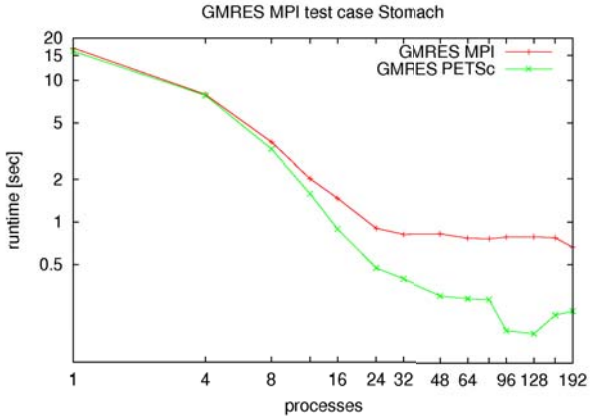


Figure 4.32. Results for the test case Stomach with four processes per IC1 node. Selected gather is used.

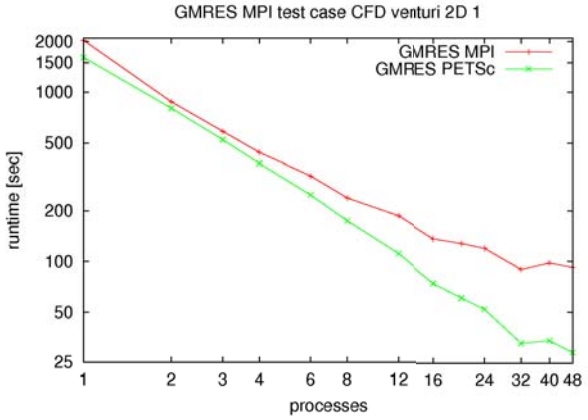


Figure 4.33. Results for the test case CFD venturi 2D 1 with one process per IC1 node. Selected gather is used.

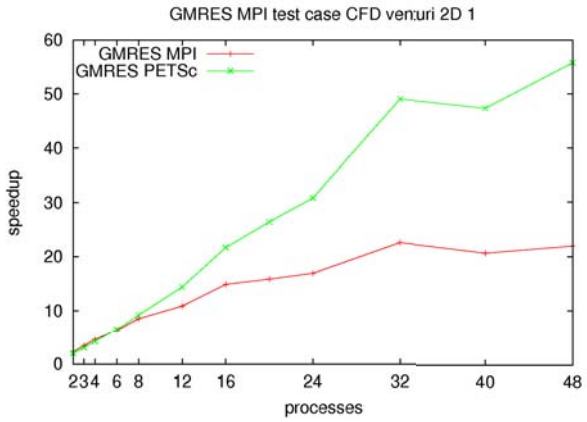


Figure 4.34. Speedups for the test case CFD Venturi 2D 1 with one process per IC1 node.

4.7.3. CUDA-Parallelization. The results for the CUDA implementation are shown in Table 4.12. Speedup values are with respect to the sequential results on the CPU for the related test case. For CFD Venturi 2D 2 and the 3D case, sequential results could not be obtained.

Test-case	Runtime [sec]	Speedup
CFD Venturi 2D 1	317,34	5,07
CFD Venturi 2D 3	3414,16	-
CFD Venturi 3D	496,69	-
Stomach	3,70	3,94

Table 4.12. Benchmark results for CUDA GMRES(m). Where no speedup is shown it was not possible to get sequential results on the CPU.

4.8. Result Interpretation

OpenMP based implementations scale well for problems that fit into the cache as the Laplace problem with 90.000 unknown shows. If the amount of data is too big for the cache OpenMP-parallelizations usually do nearly not scale any more for more than 3 threads due to limitations of memory bandwidth (the data can then be computed faster then they are delivered from the memory). Other effects arising from hardware characteristics can be found. In Figure 4.4 a trip between four and five threads can be observed, which is caused by the two-socket architecture of the Institutsclusters nodes. In the situation of four threads the program runs on a single quad-core processor, while in the case of five (or more) threads the program runs on both quad-core processors. In all test-cases, except Laplace with 90.000 unknowns, a performance limitation by the memory bandwidth on a IC1 node can be observed. Furthermore, there is no difference in the performance between the loop parallelized and the data distributed implementation. One conclusion from the presented experiments is that for (probably existing) serial code OpenMP-parallelizations or the usage of multi-threaded libraries give some additional performance on multicore platforms.

In most cases MPI-based implementations scale almost like the OpenMP-based parallelizations but for problems where the main memory is nearly exhausted the

overhead of explicit messages using MPI can become a problem. On the other hand MPI based code can easily use multiple nodes. Developing MPI code is often more complicated than employing OpenMP but on most cluster systems nowadays the performance of distributed memory nodes clearly outperforms the performance of shared memory nodes.

Hybrid implementations give no benefit for the chosen test-cases and seem not very meaningful being used.

Using coprocessor technologies like GPUs can be very beneficial in terms of performance, as the presented results demonstrate. Because of portability issues the usage of OpenCL might be interesting and could be a step for future work.

The presented CG implementation works well for the test cases Laplace 2D, CFD ACUSIM and CFD Rothberg, while showing bad performance for the test-case Thermal 2 which is a worst case scenario for the implementation. This is because the load balancing and communication patterns are bad. For the vector exchange belonging to the matrix vector multiplication the selected gather communication is used in all test cases except for Laplace 2D. In Laplace 2D neighbor communication is used to exploit the matrix structure.

More experiments should be performed to evaluate for the MPI-based solvers how different numbers of processes running on the nodes influence the performance and accuracy on different machines. These experiments should go hand in hand with analyzing the energy efficiency of the implemented solvers for certain problem classes.

4.9. Preconditioning in Parallel Solvers

There are many different types of preconditioners that can be used in combination with (parallel) CG or GMRES solvers. In the following numerical results for two frequently used preconditioners, namely Incomplete LU (ILU) and (Block-) Jacobi (BJ respectively J) and their combinations are presented.

4.9.1. Incomplete LU Preconditioning. To reduce the number of iterations of the iterative solver in a significant way is one main goal of a preconditioner. In Table 4.13 the results of a ILU(0) preconditioner with no fill-in in combination with a CG solver are presented. This combination is applied to the Laplace 2D test-case and a factor of about two between the number of iteration steps can be seen. An additional factor of about two compared to the ILU(0) can be achieved by allowing a certain fill-in (ILU_T). Since the reference examples are very small and the pure CG converges fast, the overhead to compute the preconditioner is significantly high and no performance gain could be achieved. Therefore, performance results are not presented and the author refers to the next section where ILU-preconditioning is combined with Block-Jacobi approaches. All numerical results based on the ILU-decomposition have been performed by using the *ILU++* library [**May, May07**].

Problem size	CG	CG + ILU(0)	CG + ILU_T
90.000	658	338	127
250.000	1,093	540	216
562.500	1,632	789	414

Table 4.13. CG iteration steps without, with ILU(0) and ILU_T preconditioning for test case Laplace 2D with different number of unknowns. The threshold parameter T was set to 10^{-2}.

In general it can be observed that building the preconditioner is often more expensive then to perform a loop of the pure solver. Efforts to accelerate a solver should take this fact into account. To outsource the preconditioner to a appropriate accelerator is one feasible approach.

4.9.2. Jacobi and Block-Jacobi Preconditioning. In the following, Block-Jacobi preconditioners with the MPI-based CG implementation are evaluated for the test cases CFD ACUSIM and CFD Rothberg. For the diagonal blocks the default ILU++ configuration 1010 is used performing a multilevel ILU preconditioner using pivoting by columns (to avoid small pivots) and row permutation to reduce fill-in [**ilu**].

From Table 4.13 and 4.15 it is visible that the number of iterations for the pure CG is not constant when the resources (processes) are scaled. A slight decrease is visible which can be explained by the results from Chapter 6 where it is shown that a higher grade of parallelization can lead to more accurate results and so to a faster convergence of the solver.

That there is no effect from Jacobi preconditioning for symmetric, positive define matrices when the diagonal is constant can be seen in Table 4.15. This effect is caused by the fact that on the diagonal, all entries are equal to one and is a special case of the proposition from Chapter 1.5.1.

	CG				
# Processes	no precond.	Jacobi -	Block-Jacobi 1 Block	Block-Jacobi 5 Blocks	Block-Jacobi 10 Blocks
1	2466	777	5	85	117
2	2461	778	82	118	138
3	2458	778	72	122	159
4	2462	778	80	135	174
6	2456	780	98	159	195
8	2458	780	99	176	214
12	2447	776	114	195	249
16	2451	778	134	224	274
20	2451	777	135	240	289

Table 4.14. Iterations steps of CG using different preconditioners for test-case CFD ACUSIM. The number of blocks is stated for every process and within each block ILU++ in configuration 1010 was used.

Figure 4.39 and 4.43 show the runtime results without the preconditioner setup time. Runtime results in Figure 4.40 and 4.42 include the preconditioner setup time for the Block-Jacobi benchmarks. The setup times for the plain Jacobi preconditioner are negligible. The speedup results for test-case CFD ACUSIM in Figure 4.41 are with respect to the sequential Jacobi preconditioned CG solver. In test-case CFD Rothberg the speedups are with respect to the sequential CG solver.

# Processes	CG				
	no precond.	Jacobi -	Block-Jacobi 10 Block	Block-Jacobi 50 Blocks	Block-Jacobi 100 Blocks
1	8193	8193	1299	1103	1569
2	8190	8190	726	1567	1829
3	8185	8185	837	1657	1875
4	8182	8182	1000	1818	1963
6	8184	8184	1202	1884	2235
8	8182	8182	1426	1986	2219
12	8173	8173	1613	2188	2581
16	8183	8183	1751	2192	2685
20	8183	8183	1818	2401	3039
24	8172	8172	2005	2498	3292
32	8183	8183	1981	2735	3798
40	8172	8172	1954	3026	4019
48	8179	8179	2176	3278	4440

Table 4.15. Number of iterations of a CG solver in combination with different preconditioners for the test-case CFD Rothberg. The number of blocks is stated for every process and to each block ILU++ in configuration 1010 was applied.

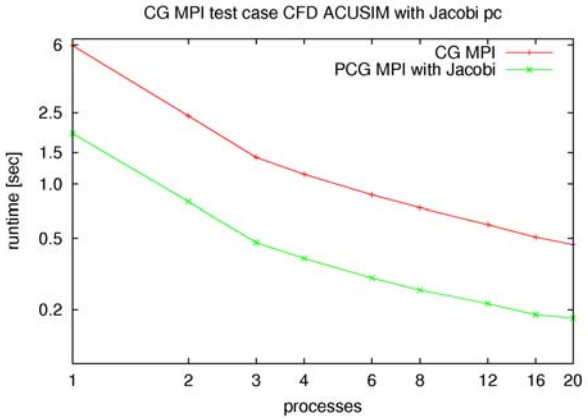


Figure 4.35. Comparison of a non-preconditioned MPI solver and a Jacobi-preconditioned solver for the test-case CFD ACUSIM.

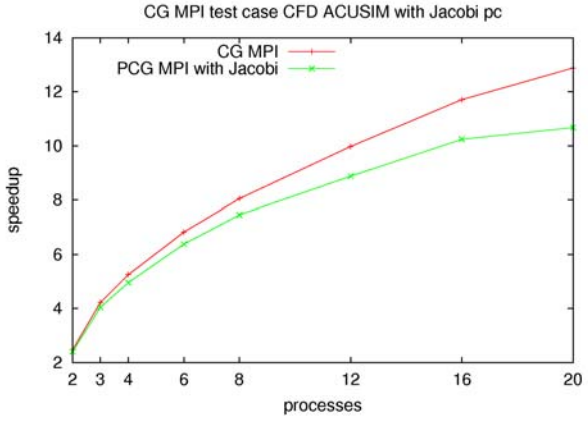


Figure 4.36. Speedups according to the results in 4.35.

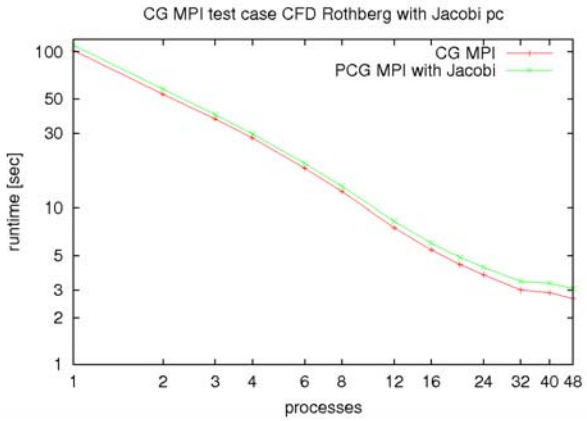


Figure 4.37. Comparison of a non-preconditioned MPI solver and a Jacobi-preconditioned solver for the test case CFD Rothberg.

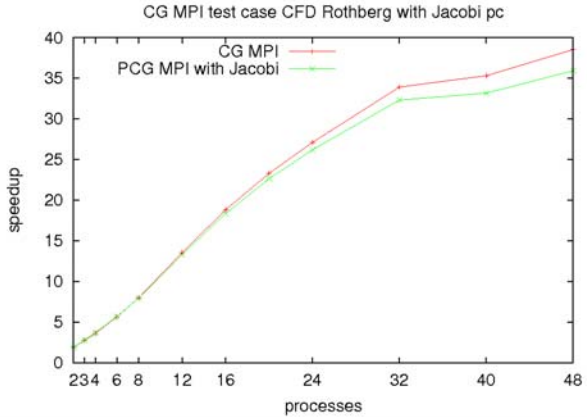


Figure 4.38. Speedups according to the results in 4.37.

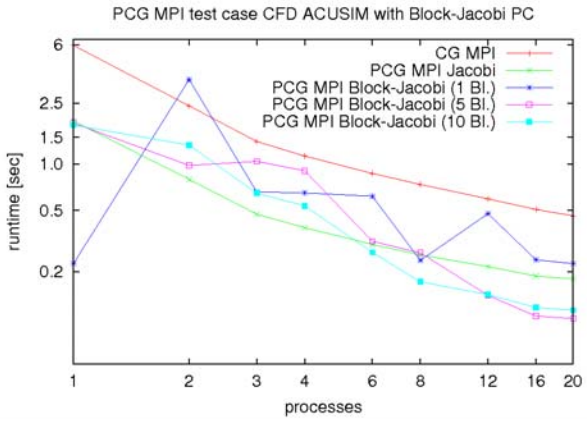


Figure 4.39. Runtime results of the unpreconditioned MPI solver, different Block-Jacobi preconditioned solvers and the Jacobi preconditioned solver for the test case CFD ACUSIM. The Block-Jacobi preconditioned solvers use one, five and ten blocks per process. Furthermore, the preconditioner setup time is not included.

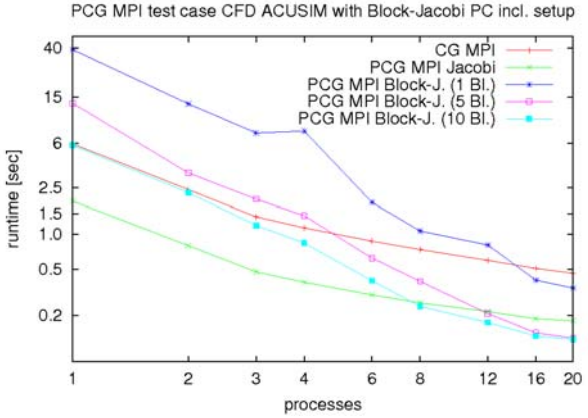


Figure 4.40. Runtime results including preconditioner setup time of the unpreconditioned MPI solver, different Block-Jacobi preconditioned solvers and the Jacobi preconditioned solver for the test case CFD ACUSIM. The Block-Jacobi preconditioned solvers use one, five and ten blocks per process.

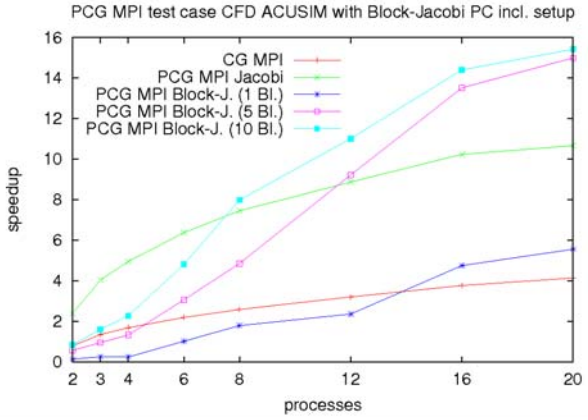


Figure 4.41. Speedup results of the unpreconditioned MPI solver, different Block-Jacobi preconditioned solvers and the Jacobi preconditioned solver for the test case CFD ACUSIM. The Block-Jacobi preconditioned solvers use one, five and ten blocks per process.

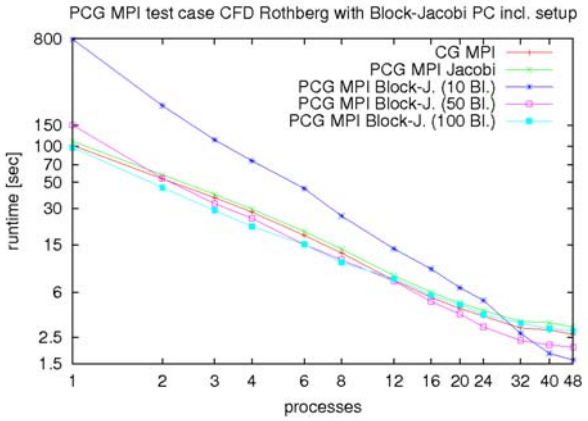


Figure 4.42. Runtime results including preconditioner setup time of the unpreconditioned MPI solver, different Block-Jacobi preconditioned solvers and the Jacobi preconditioned solver for the test case CFD Rothberg. The Block-Jacobi preconditioned solvers use 10, 50 and 100 blocks per process.

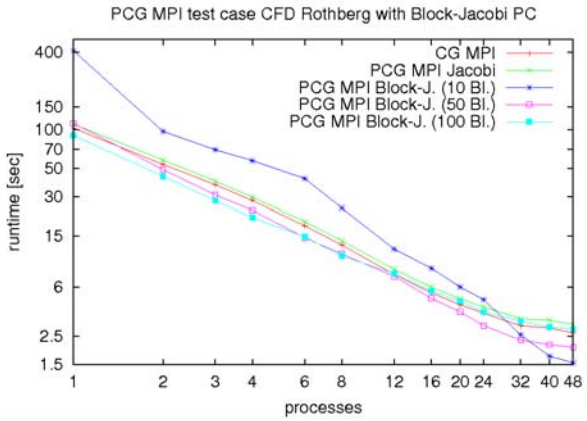


Figure 4.43. Runtime results of the unpreconditioned MPI solver, different Block-Jacobi preconditioned solvers and the Jacobi preconditioned solver for the test case CFD Rothberg. The Block-Jacobi preconditioned solvers use 10, 50 and 100 blocks per process. Furthermore, the preconditioner setup time is not included

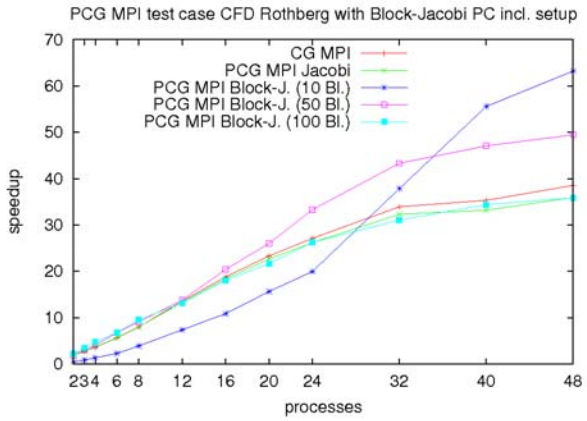


Figure 4.44. Speedup results of the unpreconditioned MPI solver, different Block-Jacobi preconditioned solvers and the Jacobi preconditioned solver for the test case CFD Rothberg. The Block-Jacobi preconditioned solvers use 10, 50 and 100 blocks per process.

The presented results indicate that there is a tradeoff between preconditioner setup time and the acceleration of the solver by the preconditioner. Block-Jacobi preconditioners with good acceleration properties often show high setup times compared to the solver time. In contrast, the Block-Jacobi preconditioners with faster setup times lead to more iteration steps of the iterative solver. The smaller the dimension of the blocks used by the Block-Jacobi preconditioner the faster is the setup time and therefore users should choose the right combination of blocks used in the preconditioner, factorization of the blocks and also the level of parallelization according to their application.

CHAPTER 5

Prototypical Evaluation of Hardware Capabilities

Essential for an efficient usage of modern hardware resources is a deep understanding of the available capabilities. Such can be very different from machine to machine and are depending on the hardware components, e.g. architecture of the processor(s), memory, chip-set, accelerators, interconnect and other involved components.

This chapter shows prototypical approaches for evaluating cluster systems (Chapter 5.1), as well as systems that are equipped with GPUs (Chapter 5.2) or FPGAs (Chapter 5.3). Finally, an inter-architectural comparison in terms of performance is shown (Chapter 5.4) based on the iterative refinement method.

Parts of this chapter have already been reviewed, presented and published on the SIAM-Conference on Parallel Processing for Scientific Computing (Seattle, Washington February 2010), the PARS workshop (arranged by the Gesellschaft für Informatik; Parsberg, Germany June 2009) and the EMCL-preprint series. Associated publications are [**AHHRed**], [**ARH10**] and [**HRR09**].

5.1. Cluster Computing

Most cluster systems used today for high-performance scientific computing are built from off-the-shelf standard components placed in racks. Usually this leads to predictable results with respect to the available performance based on similar hardware. When more exotic hardware shall be tested, as many information as possible should be gathered in order to tailor numerical methods and implementations as good as possible for the system.

SiCortex has chosen a strategy not to use standard components and offers a line of integrated cluster machines based on a customized low-frequency MIPS multicore processor and a specialized network fabric.

Investigated is the potential of the SiCortex platform for numerical simulation by analyzing the performance of a set of elementary benchmarks and two fluid dynamics applications executed on the SC072 and the SC5832 systems. The elementary benchmarks quantify the performance in terms of computation rate, memory bandwidth and communication latency. The fluid dynamics applications provide insight into how well existing scientific code performs on the system. The results are compared to those obtained on a commodity cluster with Intel Xeon cores and Infiniband interconnect. The focus of the evaluation is computational performance, but also the energy consumption for all three machines is considered.

The presented results indicate that while the SiCortex systems might be well suitable for applications that can be parallelized to a very fine level, they are outperformed by commodity clusters when this is not the case. However, an analysis of the CFD applications shows that the SiCortex systems make it possible to significantly reduce the energy consumption compared to a commodity cluster.

During the past years systems composed of off-the-shelf components have dominated the market for high performance computing clusters. A company building machines in a different way is SiCortex with the products SC072 and SC5832. These systems are based on low-frequency MIPS64 cores and use a customized interconnect for communication. The idea is to provide a machine that better balances the computation rate of the CPU and the memory bandwidth by spreading the computations over a very large number of cores. For applications that can be parallelized with a fine granularity, the low-power nodes are thought to have the potential to also reduce the energy costs of large scientific computations.

This chapter presents the results of several benchmarks performed on the SC072, the SC5832, and a commodity cluster based on Intel Xeon cores and Infiniband interconnect. The benchmarks include both elementary kernels and two full applications that solve a problem in fluid dynamics.

A description of the hardware and software used for the tests is given in Sections 5.1.1 and 5.1.2. The results of the benchmarks are presented in Sections 5.1.3 and 5.1.4. Section 5.1.5 summarizes the analysis of these results.

5.1.1. Hardware Description. This section gives an overview of the three cluster systems used in the tests.

The *SiCortex SC5832* is a single cabinet machine with 36 circuit modules each accommodating 27 SiCortex node chips and associated memory [**SiC**]. Each node hosts a six-core MIPS64 processor with a clock rate of 700 MHz. The most important specifications of the machine used for the test are listed in Table 5.1.

All nodes on the SC5832 can communicate via a customized interconnect, which is based on a Kautz-graph topology of degree 3 and diameter 6. In this network topology, there are 3 disjoint paths from each node to any other node, which provides fault tolerance in the case that a node or link fails. A maximum of 6 hops are required to transmit a message between any two nodes.

The *SiCortex SC072* is the desktop version of the bigger SiCortex machine and contains 12 nodes with 72 cores in total. The diameter of the Kautz-graph with degree 3 is 2 for this machine. The machine used for the test is equipped with 8 GB of main memory per node. A performance analysis of an earlier version of the SC072 is available in [**MLID09**].

The commodity cluster used for the comparison is the *Institutscluster IC1* located at the Steinbuch Centre for Computing (SCC) [**fC**] at the former University of Karlsruhe, now Karlsruhe Institute for Technology (KIT) [**oT**]. It consists of five cabinets containing a total of 200 computing nodes, each equipped with two Intel quadcore EM64T Xeon 5355 processors with Clovertown architecture running at 2.667 GHz. The interconnect used is Infiniband 4x DDR.

Table 5.1 summarizes the characteristics of these three systems. Most values come from the specifications given by the respective manufacturers. The values for the power consumption come from three sources: for the SC072, the power was measured with a wattmeter by the authors; for the SC5832, the power was taken from a webpage of the University of Magdeburg [**Sch**]; and for the IC1, the values were communicated from the SCC. The values of the Linpack benchmark computation rate come from own measurements (SC072), the HPCC results database (SC5832) and the website of the SCC (IC1).

5.1.2. Software Description. The benchmarks that were used fall into two categories: elementary kernel benchmarks, which aim to measure an isolated aspect

	SC072	SC5832[a]	IC1[b]
Nodes	12	972	200
Processors per node	1	1	2
Cores per processor	6	6	4
Th. comp. rate / core	1.4 GFlop/s	1.4 GFlop/s	10.7 GFlop/s
Th. comp. rate / node	8.4 GFlop/s	8.4 GFlop/s	85.3 GFlop/s
Th. comp. rate full machine	100.8 GFlop/s	8.2 TFlop/s	17.6 TFlop/s
Linpack full machine	56.6 GFlop/s	4.73 TFlop/s[c]	15.2 TFlop/s
L2-cache per processor	1.5 MB	1.5 MB	8 MB
Memory per node	8 GB	4 GB	16 GB
Memory full machine	92 GB	4 TB	32 TB
Memory bandwidth per node	10.7 GB/s	10.7 GB/s	10.7 GB/s
Power consumption load	330 W	21 kW	103 kW
Power consumption idle	230 W	11 kW[d]	56 kW

[a]The evaluated machine is located at the computing center of the University of Magdeburg. (See [**Sch**])
[b]See http://www.rz.uni-karlsruhe.de/ssck/ic.php
[c]See [**HPC**]
[d]See [**Sch**]

Table 5.1. Key system characteristics of the three clusters used for the tests.

of the performance of the machine, such as its floating point computation rate, its memory bandwidth or its interconnect communication speed; and a computational fluid dynamics (CFD) application benchmark, which gives an indication of how well the systems perform on typical scientific simulations. This section presents an overview of the software used to test the machines.

On the IC1 the Intel compiler version 10.1.022 was used together with the Intel MKL 10.1 numerical library. On the SiCortex machines the Pathscale compiler and ATLAS BLAS 3.7.32 for the benchmarks is used.

5.1.2.1. *Elementary Benchmarks.* **HPCC Suite:** The HPC Challenge Benchmark is a suite of seven benchmark kernels created by the DARPA HPCS program [**DL05**]. Together they provide a more balanced view of the performance of general-purpose HPC systems than the classical Linpack benchmark, whose performance is limited mainly by the rate of floating point arithmetic that the processors are capable of. In the present work, the focus lies on two of the benchmarks: the MPI ping-pong latency and the STREAM *triad* tests. Full benchmark results for the SC5832 are available in the HPCC results database [**HPC**]. **LLCbench:** The LLCbench benchmark collection consists of three parts, aiming to measure the performance of simple linear algebra functions (Blasbench, [**MLT98**]), the performance of the memory hierarchy (Cachebench, [**MLT99**]) and the speed of the communications interconnect (MPbench). This report includes results obtained with the first two of these benchmarks.

5.1.2.2. *Application Benchmarks.* The numerical simulation test case is a standard example in fluid dynamics: 3D stationary lid-driven cavity on a cuboid. This problem is solved with the finite element software HiFlow and the Lattice-Boltzmann software OpenLB.

HiFlow: The HiFlow package is a parallel, finite element library with a strong emphasis on computational fluid dynamics and is written in C++. The library includes the following features:

- CFD (incompressible Navier-Stokes, Low-Mach flows, heat convection)
- Reactive flows
- Eigenvalue computation for stability analysis
- Conforming h- and hp-FEM
- A posteriori error estimation for FEM
- Moving boundaries

HiFlow uses the PETSc [**BBE+08, BBG+09, BGMS97**] and METIS [**KK99**] libraries for solving this linear system in parallel on all three machines. Meanwhile the HiFlow package is available as Open Source project under the name HiFlow^3 [**AAB+10**] while the shown benchmarks have been performed with the predecessor of HiFlow^3.

OpenLB: The OpenLB project [**Ope**] provides a publicly available C++ library mainly intended for researchers and engineers who simulate fluid flows by means of lattice Boltzmann methods [**LBM**]. The library includes the following features:

- Flows in complex geometries
- Turbulent flows
- Multiphase flows
- Thermal flows

The library enables a fast implementation of both simple applications and advanced CFD problems. It is easily extensible to take into account new physical models.

The used version of OpenLB is capable of hybrid parallelization using MPI for inter-nodal communication and OpenMP within the nodes. For a deeper analysis of the parallelization see [**HKL09**].

5.1.3. Elementary Kernels Performance Results.

5.1.3.1. *HPCC MPI-latency Results.* The MPI latency benchmark from the HPCC benchmark suite measures the latency of the communication between two cores by sending messages from one core to another. The message size for the test is 8 bytes. All pairs of cores are used in the test, and the minimum, maximum and average values of the latency are reported. The test was run using different number of nodes, with either one or all cores per nodes active, and the results are shown in Fig. 5.1.

The spread of the latency on the SC072 machine is very small due to the fact that either one or two hops are required for communication between any two cores. On the SC5832 where the topology has diameter 6, the spread is a little larger. Much larger fluctuations are seen on the IC1, which has a fat-tree topology. [**GS05**] provides more information on network topologies and their impact on communication performance.

When all cores per node are used, the IC1 has a lower minimum latency than the SiCortex machines. A comparison with the test with one core per node shows that this minimum corresponds to communication within a single node. The average latency on the IC1 is circa 2 μs with 4 nodes and almost 3 μs with 9 nodes. On the SiCortex machines, the average latency with both 12 and 72 nodes is less than 1.5 μs, which indicates a better scalability with respect to this metric than the IC1.

5.1.3.2. *Cachebench Results.* On each of the three clusters the Cachebench benchmark was run on a single core. Results for the SC5832 and the IC1 are shown in Fig. 5.2 and Fig. 5.3. The corresponding graph for the SC072 has been omitted since the results were practically identical to those of the SC5832.

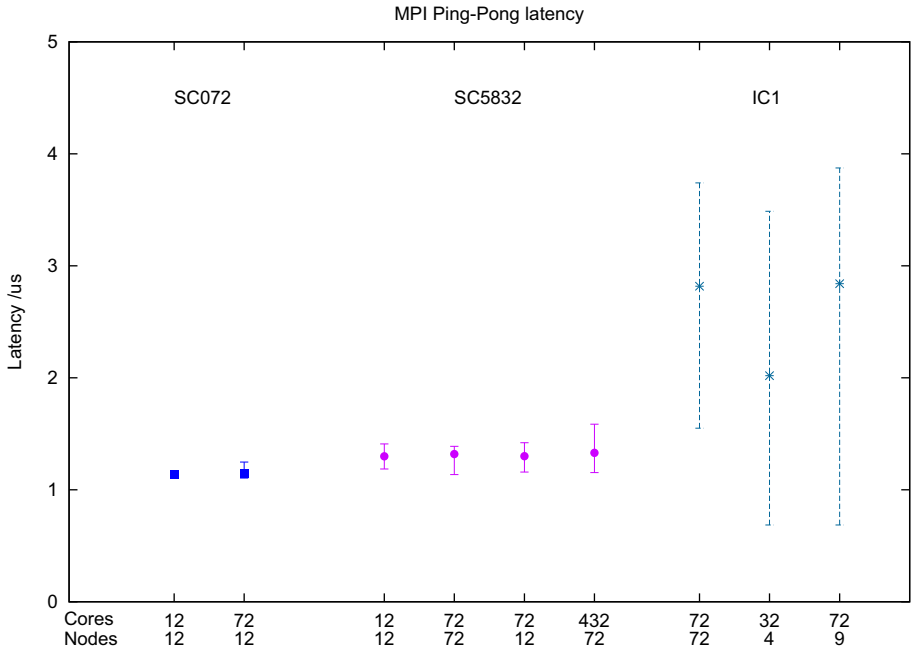


Figure 5.1. MPI Ping-Pong latency. The graph shows the average and the variation around this value.

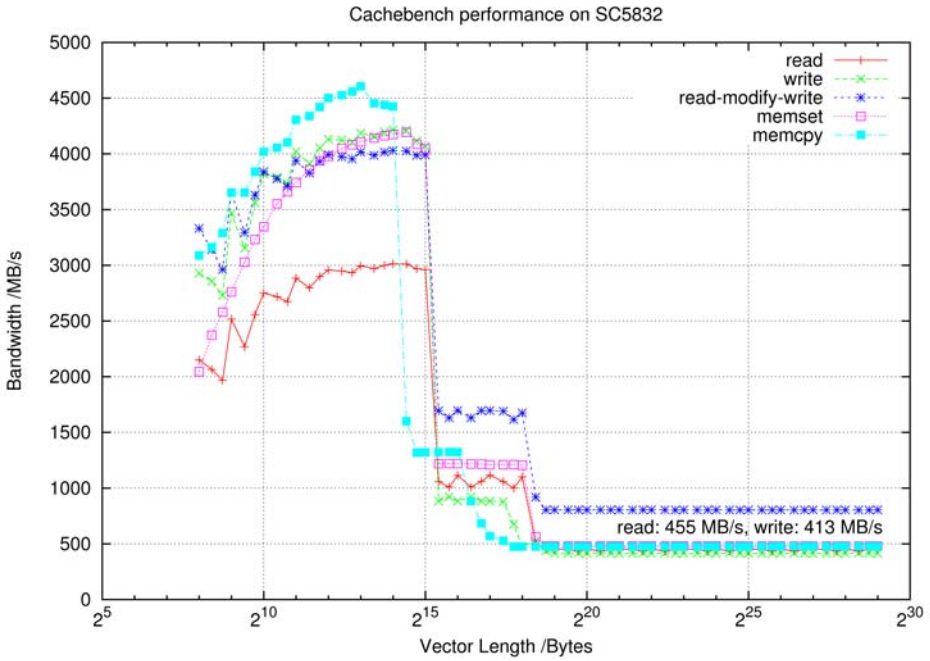


Figure 5.2. Performance of memory hierarchy on the SC5832 which are similar for the SC072.

The effect of the use of cache memory on both machines is obvious from the graphs. The L1-cache of both processors has the same size (32 kB) but the bandwidth on the Xeon processor is approximately 8 to 10 times higher on the read and write operations. The L2-cache, which in contrast to the L1-cache is not exclusive for each

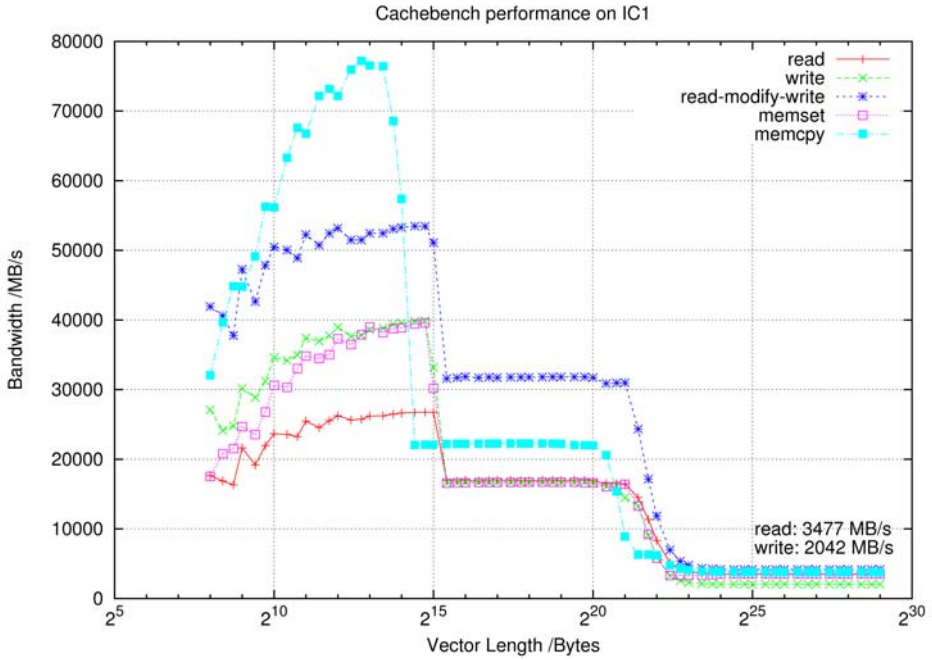


Figure 5.3. Performance of memory hierarchy on the IC1.

core, is considerably larger on the Xeon processors. It can be shared dynamically on the Intel processor, but not on the SiCortex, which leads to a steeper decrease in bandwidth on the latter system when the data cannot be held in this cache memory.

For arrays which fit into the L1 cache on the IC1, the *memset* and *memcpy* operations perform considerably better than the hand-coded write and read-modify operations; while on the SC5832, the difference in performance is minor. Clearly, the system calls have been tuned to take advantage of the special features of the Intel architecture, while the same is not true for the SiCortex MIPS platform. For large arrays, the hand-coded versions are equivalent or better than the library calls.

For applications that intensively access the memory, the most important information that can be extracted from the graphs is the bandwidth to the main memory, which can be read at the far right of each graph. The bandwidth on the IC1 is approximately 7.5 times higher than that of the SC5832 for the read operation, and 4.9 times higher for the write operation.

5.1.3.3. *HPCC STREAM Results.* The STREAM benchmark evaluates the memory performance within one node by performing a vector operation on an array of data that does not fit into the L2 cache. The test has two different modes: "single" mode, where the test is run only on one randomly chosen core; and "star" mode, where all active cores run the test. The star mode makes it possible to measure the degradation of the memory bandwidth when all cores on a node require access to the memory controller simultaneously.

The graph in Fig. 5.5 shows the result of the *triad* ($z = \alpha x + y$) operation for different numbers of nodes. In each configuration either one or all cores were active on each node. When only one core is active on each node, the single and star modes give the same results, as one would expect.

The bandwidths achieved in single mode correspond to those measured with the Cachebench benchmark. The results for the *copy* operation are similar to those

of the *triad* since the performance of both operations are limited by the memory bandwidth and not by the computational rate.

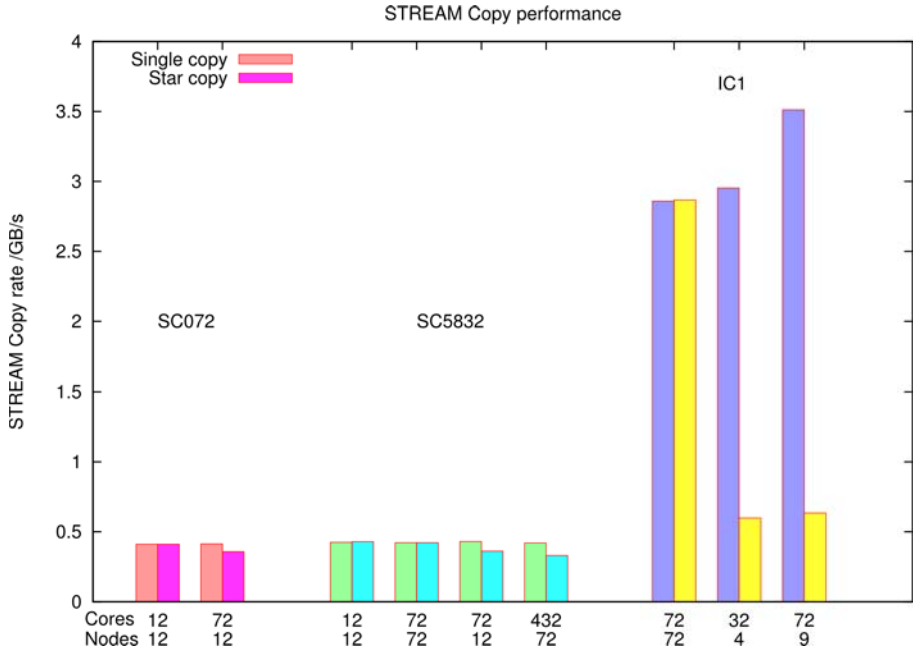


Figure 5.4. Single- and Star-STREAM copy performance for SC072, SC5832 and IC1.

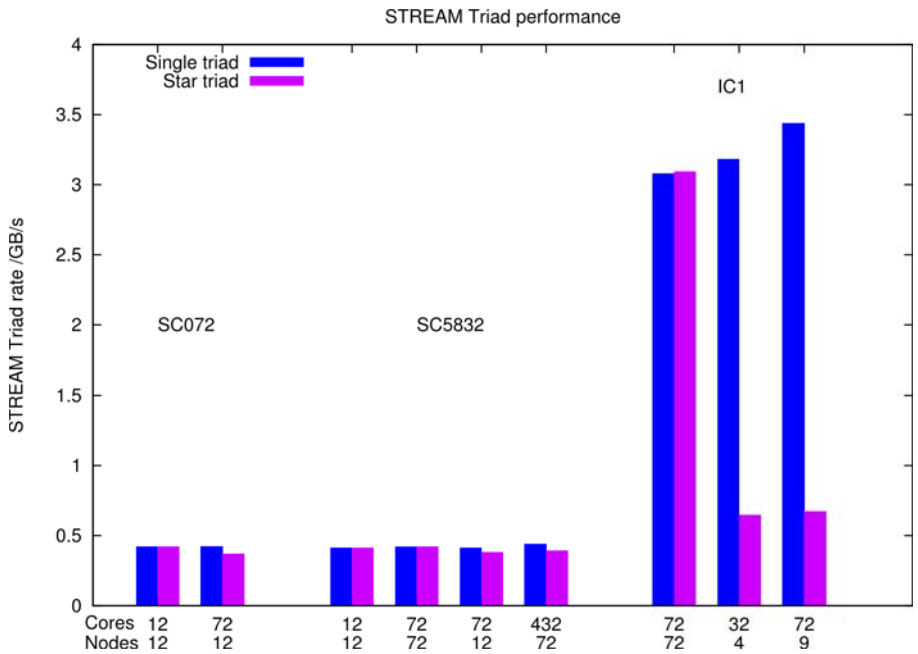


Figure 5.5. Single- and Star-STREAM triad performance for SC072, SC5832 and IC1.

On the IC1 cluster, the measured bandwidth is reduced drastically when going from the single mode to the star mode, whereas on the SC072 and the SC5832

this reduction is much less pronounced. This suggests that the memory bandwidth is less likely to be a bottleneck when scaling an application from one to six cores per node on the SiCortex systems than on the IC1. On the latter system, the time of execution of a parallel application can often be reduced by spreading the processes over more nodes and using fewer cores on each node, in order to increase the available memory bandwidth. The results of the STREAM benchmark suggest that the same is not likely to be true on the SiCortex machine.

However, it should not be forgotten that even when all cores are used, the measured bandwidth is 50% higher on the IC1 than on the SiCortex machines.

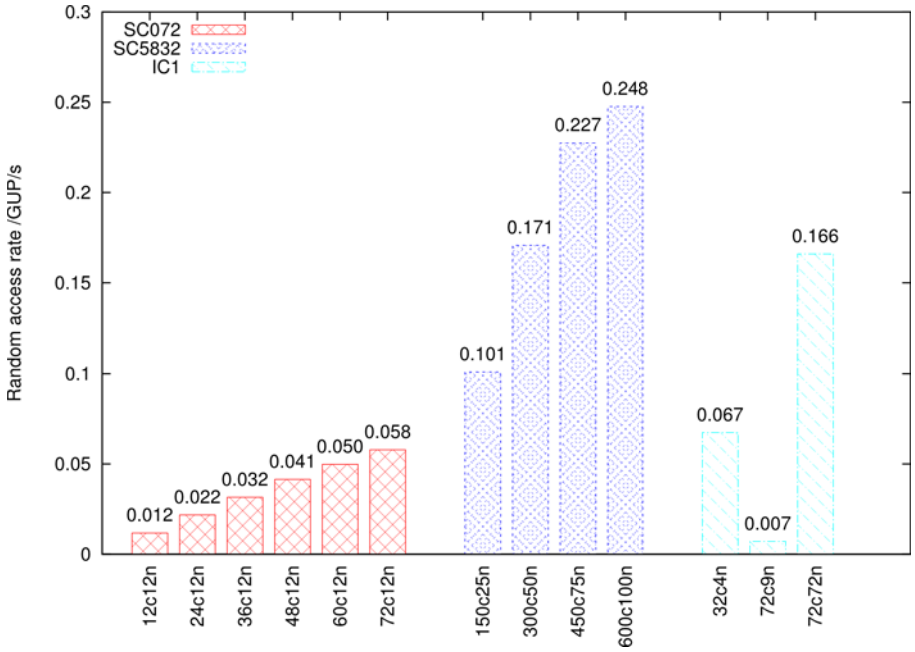


Figure 5.6. Random access rate in GUP/s.

5.1.3.4. *Blasbench Results.* The Basic Linear Algebra Subroutines (BLAS) are a central part of many scientific computing codes. The Blasbench test-suite measures the single-core computation rate of a kernel from each of the three levels of the BLAS. These kernels are *daxpy*, *dgemv*, and *dgemm*, which respectively compute vector-vector, matrix-vector and matrix-matrix multiplications.

Figures 5.7 and Figurefig:8b show the results of the Blasbench test on the SC5832 and IC1, respectively. The graph for the SC072 is omitted since it is identical to that of the SC5832.

All three kernels show an increase in computation rate with the characteristic size of the problem N up to a certain limit, where the curves flatten out. On the IC1, the performance is higher for problem sizes up to $N = 1024$, since the data can fit into the L2 cache. On the SC5832, this effect is absent, which suggests that the ATLAS BLAS implementation does not make efficient use of the cache.

Overall, the *daxpy* operation is over 22 times faster on the IC1 than on the SiCortex machines, and the *dgemv* operation is 8 times faster. For *dgemm*, the IC1 is 9.5 times faster, which should be compared to the ratio 7.6 in theoretical peak computation rate between the two platforms. The IC1 achieves circa 90% of its peak performance with the *dgemm* kernel, while the SiCortex systems reaches only about 70% efficiency. It seems clear that the highly optimized Intel MKL implementation

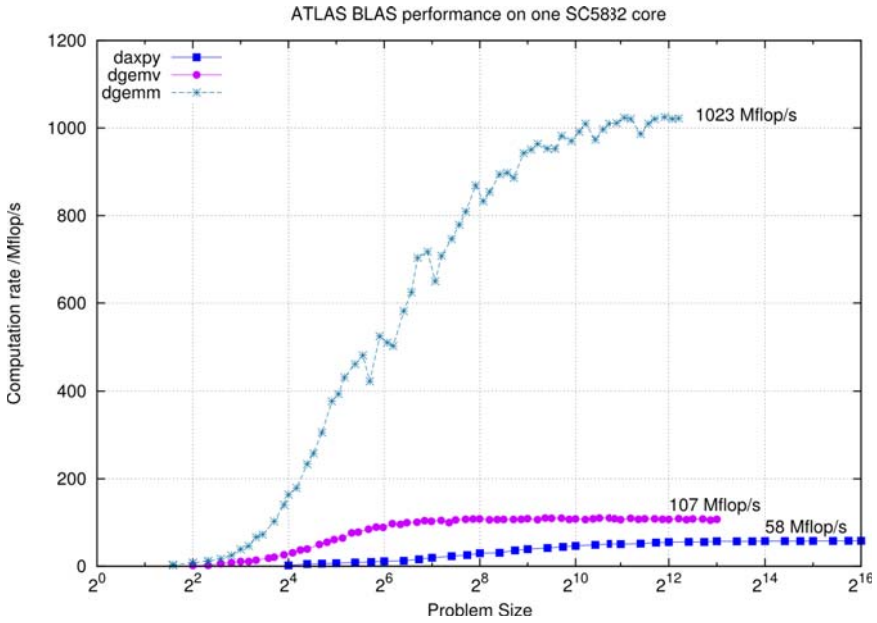


Figure 5.7. Single core BLAS performance on the SC5832. The performance on the SC072 and the SC5832 is identical.

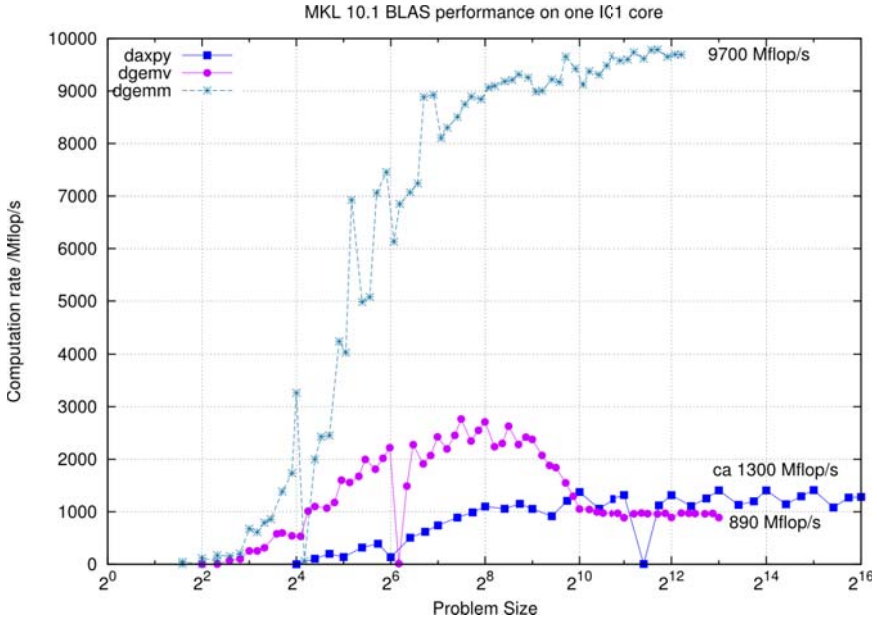


Figure 5.8. Single core BLAS performance on the the IC1.

on the Intel processors significantly outperforms the ATLAS BLAS library on the SiCortex MIPS64 processors.

It should however be noted that the IC1 shows some irregular but reproducible variations with the problem size which are not present on the SiCortex systems.

5.1.4. Performance and Energy Efficiency of Applications.

5.1.4.1. *Test case for HiFlow.* The test case for the CFD package HiFlow is a standard example: 3D lid-driven cavity (LDC) on a cuboid. The geometry is

uniformly refined to 65536 cells and the incompressible Navier-Stokes equations are solved for a stationary solution with Q2 elements for the velocity and Q1 elements for the pressure. The number of unknowns in the resulting linear system of equations is approximately 1.7 million.

Fig. 5.9 shows the time taken to assemble and solve the lid-driven cavity problem with different number of processes distributed in different ways over the nodes on the three clusters. For all three machines, there is a point after which the gain in performance for each added process starts decreasing. This phenomenon, which is typical when a fixed problem size is used, is due to Amdahl's law, which states that the gain in speed from parallelization is limited by the fraction of time spent in sequential code.

Overall, the slope of the curves is similar for the SiCortex machines and the IC1 up to approximately 128 processes, at which point the performance increase with each added process falls off rapidly on the SiCortex systems.

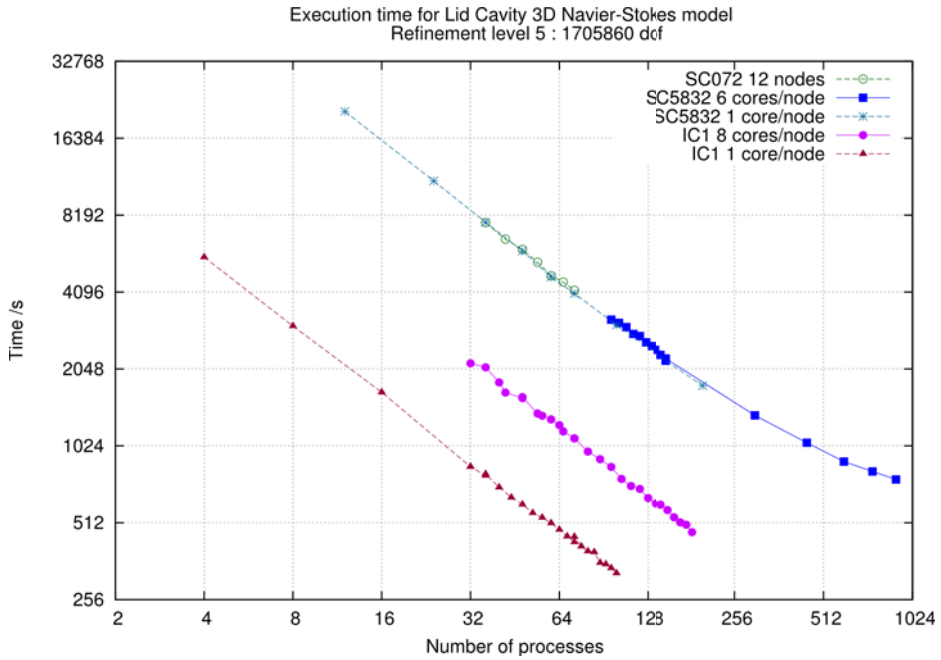


Figure 5.9. Execution time for 3D lid-driven Cavity with HiFlow on the IC1, SC072 and SC5832. Incompressible Navier Stokes equations are solved with 1,705,860 degrees of freedom.

As was seen for the STREAM benchmark in Section 5.1.3.3, this test shows the impact of the memory bandwidth limitation on the IC1 through the fact that the execution with 1 core per node is more than twice as fast as with 8 cores per node. No such difference exists on the SiCortex systems, which can be interpreted as if the SiCortex systems have a better balance between computation rate and memory bandwidth. The fact remains, however, that the execution time on the IC1 is much lower than on the SiCortex machines. As an example, with 96 processes, the IC1 solves the problem 9.4 times faster than the SC5832 with one core per node and 3.8 times faster with eight cores per node. These performance ratios are similar also with other numbers of processes.

The main promise of the SiCortex systems is not computation speed but rather efficiency in terms of energy. It is therefore interesting to investigate this aspect of

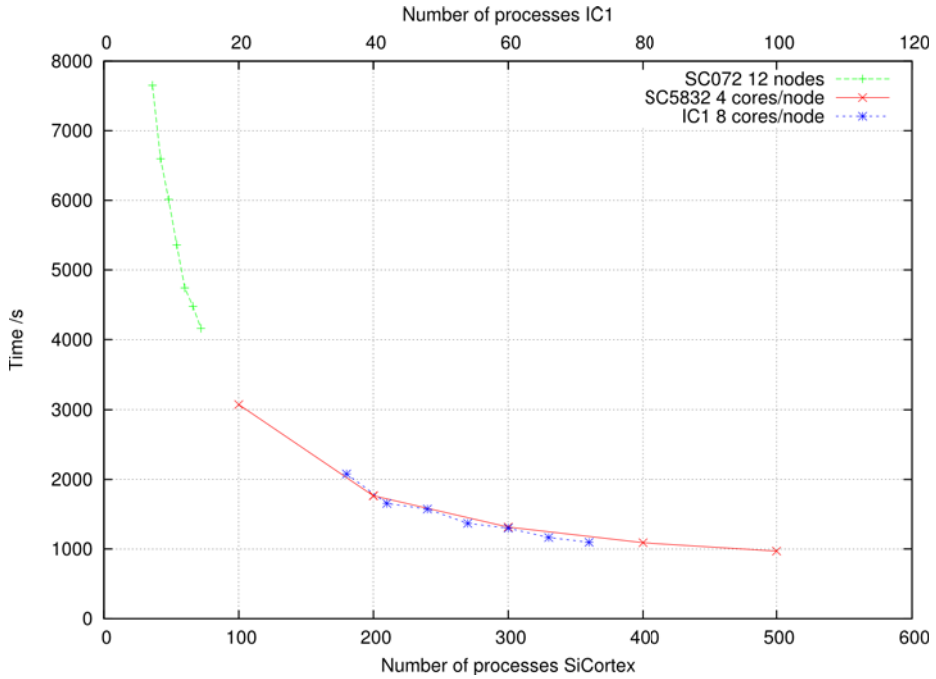


Figure 5.10. Execution time for 3D Lid Driven Cavity with HiFlow on IC1, SC072 and SC5832. Incompressible Navier-Stokes equations are solved with 1.705.860 degrees of freedom.

the cluster systems. By deducing the power consumption P under load per node from the values given in Table 5.1, 514 W per node on the IC1, 21 W on the SC5832 and 28 W on the SC072 are obtained. With these values, it is possible to estimate the total amount of energy E that is been spent for a computation which takes t seconds through the relation $E = P \cdot t$.

Fig. 5.12 shows the energy consumption as a function of the time taken to solve the problem. As long as the execution time scales perfectly with the number of cores, the curves are flat, since the increase in power from using more cores is compensated by a corresponding decrease in execution time. Using one core per node on the IC1 clearly makes it possible to solve the problem in the shortest time, but the energy cost is very high. To lower the energy consumption, all cores should be used on each node. For execution times over 1300 s, where the scaling is good on the SC5832, the energy consumption for a fixed execution time on this machine is between 3 and 4 times lower than on the IC1. Hence, if one can afford to let the computation take a longer time to finish, large energy savings are possible on the SiCortex system. Fig. 5.11 shows the energy consumption as a function of the number of processes for the different configurations. As long as the scaling is good, the increase in energy consumption from using more cores is small, because each added core brings about an almost proportional decrease in the computation time. For large numbers of processes, adding a process no longer decreases the computation time very much, and will therefore increase the energy consumption.

In absolute figures, the solution can be obtained with a much lower energy consumption on the SiCortex machine than what is possible on the IC1, despite the longer execution times. Using only one core per node is obviously very wasteful from an energy perspective, but this practice is not uncommon among users on the IC1, who want to avoid the bottleneck of the limited memory bandwidth, and obtain their results as fast as possible.

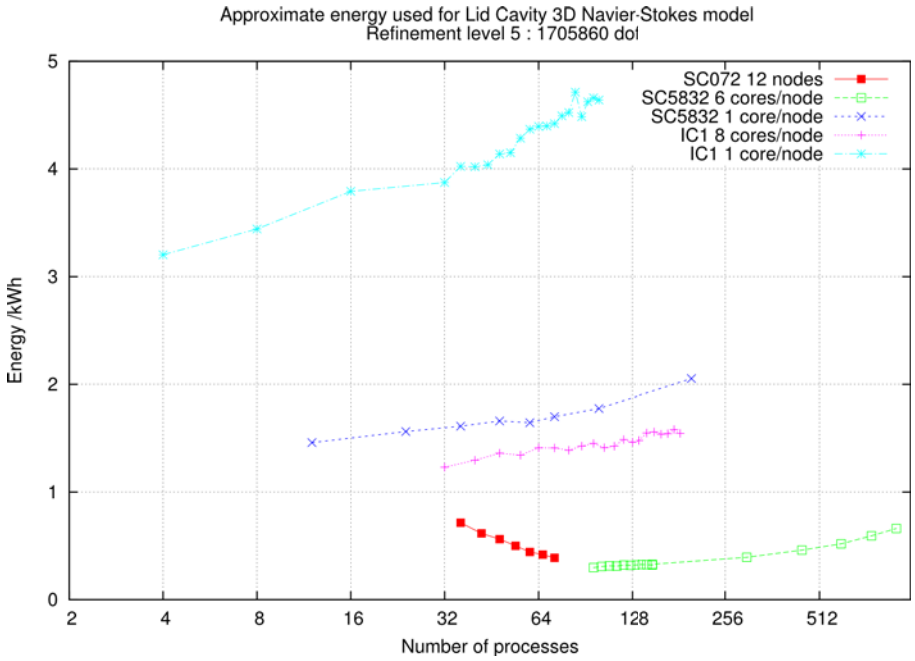


Figure 5.11. Energy consumption in relation to the utilized resources for solving the Navier-Stokes equations on a cube with finite elements. The problem size is constant with 1.7 million unknowns in the discretized linear system of equations. On IC1 and SC5832, configurations with one and all cores per node were tested; while on SC072 twelve nodes were always used, with different number of cores.

5.1.4.2. *Test case for OpenLB.* The test case for the Lattice Boltzmann/CFD package OpenLB was again the 3D lid-driven cavity simulation, this time with an instationary flow on a cubic geometry. A Lattice Boltzmann method with D3Q19 discretization model was used as explained in [**HKL09**]. In order to decrease the execution time of the tests, only the first twenty time steps were computed, although a typical simulation would use many more time steps. The execution time was therefore dominated by the time to initialize the data structures; whereas in a real computation this time would be negligible. Hence the initialization time was not taken into account for the time measurements.

The size of the problem was scaled with the number of processes used so that when the scaling is perfect, the computation time should stay constant. The problem size was chosen differently on IC1 and SC5832, in order to obtain approximately the same execution time on both systems.

Disregarding the initialization time, both the execution time and memory consumption of the program scale as $\mathcal{O}(\frac{N^3}{p})$ where N is the number of discretization points in each dimension and p the number of processes. In order to keep the execution time T constant when varying p, the following relation was used to determine the size of the problem: $N = \lfloor \alpha \sqrt[3]{p} \rfloor$. For the results shown in Fig. 5.13, $\alpha = 100$ was used on the SC5832 and $\alpha = 180$ was used on the IC1, which makes the latter problem almost 6 times bigger than the former.

The results achieved with OpenLB are representative for explicit methods which are usually limited by the available memory bandwidth. The results in Fig. 5.14 show that the computation time stays approximately constant, which indicates that this code scales well on both machines.

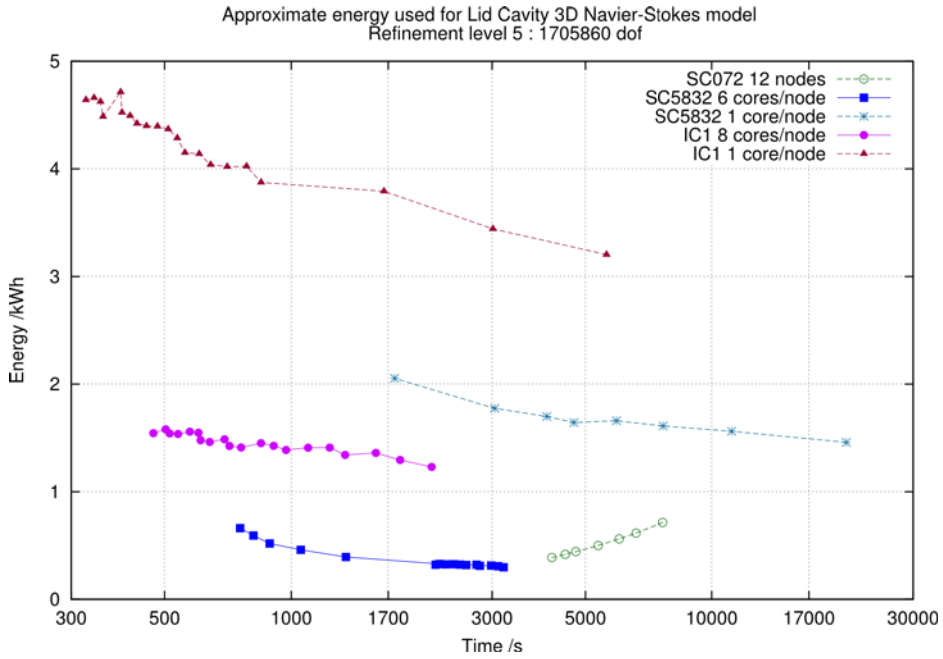


Figure 5.12. Energy consumption in relation to the computation time for solving the Navier-Stokes equations on a cube with finite elements. The problem size is constant and leads to a linear system with 1.7 million unknowns. On the IC1 and SC5832, configurations with one and all cores per node were tested; while on the SC072 twelve nodes were always used, with different number of cores per node.

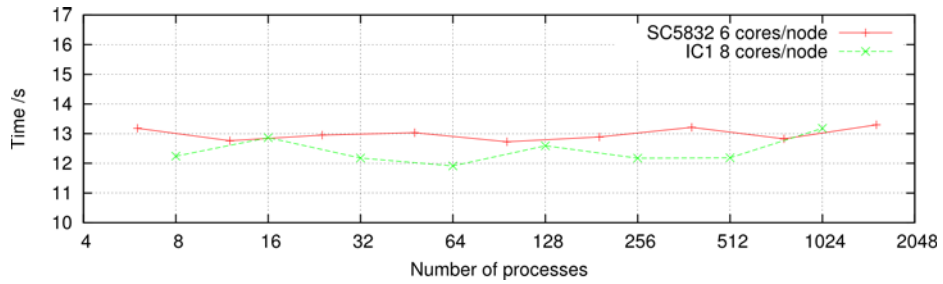


Figure 5.13. Execution time for 20 timesteps of 3D lid-driven Cavity with OpenLB on the IC1 and SC5832 without initialization time. The problem was scaled according to Figure 5.14.

5.1.5. Conclusion. The results of the benchmarks have given some insight into the characteristics of the integrated and custom-designed SiCortex SC072 and SC5832 cluster systems. Their performance has been compared to that of the IC1, which is a system assembled from off-the-shelf components from different vendors.

In terms of floating-point computation rate for each core, the SiCortex systems are clearly inferior to the IC1. Sequential parts of an application thus have a larger risk to become a limiting factor on the SiCortex system than on the commodity cluster.

The ability to fully exploit multicore processors is often limited by the bottleneck associated with access to the main memory. The HPCC STREAM results show that this bottleneck has been removed on the SiCortex systems, whereas it is very significant on the IC1. This observation is confirmed through the CFD application benchmark, where the IC1 exhibits a large difference in execution time when only

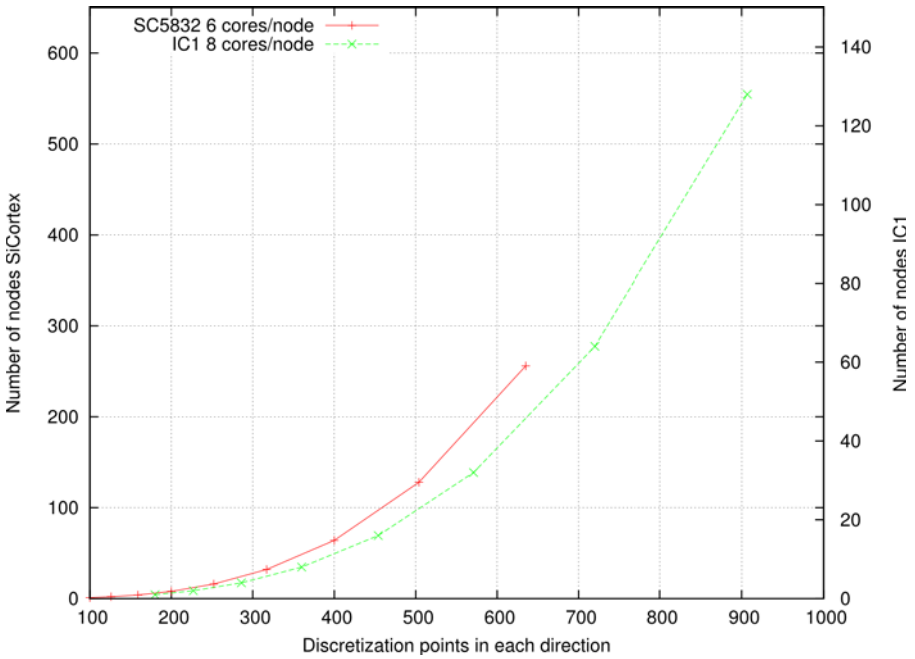


Figure 5.14. Scaling of the problem size according to $N = \lfloor \alpha \sqrt[3]{p} \rfloor$. $\alpha = 100$ on the SC5832, and $\alpha = 180$ on the IC1.

one core per node is used instead of all cores. On the SiCortex machines, all cores per node can be used without performance degradation.

On the other hand, the absolute performance of the SiCortex nodes is somewhat disappointing: the IC1 performs 50% better in the HPCC STREAM benchmark, and is 8 times faster on the BLAS *dgemv* operations. There seems to be room for improvements both of the hardware and the software.

When a full node is used, the IC1 also outperforms the SiCortex systems, but the ratio in performance is usually smaller than for a single core, even though the number of cores per node is higher on the IC1. One reason for this is that the memory bandwidth becomes a limitating factor on the IC1 to a larger extent than on the SiCortex machines. This is illustrated by the results of the HPCC STREAM benchmark, which indicate that the SiCortex cores are so slow that common scientific computations are not limited by the bandwidth of the memory.

In order to achieve comparable execution times with a given application on the slower processors in the SiCortex system, it is necessary to spread the computation over a larger number of nodes. This is made possible by the low latency of the system's network. The potential for exploiting a large number of cores is also determined by the scalability of the software. For the tested parallel fluid dynamics codes the SC5832 performs well compared to the IC1. The finite element CFD application requires 4 to 5 times more cores on the SiCortex machine than on the IC1 to achieve the same computation time, when all cores per node are used. The Lattice-Boltzmann approach also scales very well.

The SiCortex machines can also be used in a more energy efficient way than the x86 cluster. For a fixed computation time, an execution of the finite element application on the commodity cluster consumes between 3 and 4 times more energy, when all cores per node are used. In view of current environmental concerns, this makes the SiCortex platform a very interesting alternative for tasks that are not time-critical.

5.2. GPU-Accelerated Scientific Computing

Recently, the number of users and lines of code taking advantage of the computational power of accelerators, especially GPUs, grew enormously. One reason is the facilitated programmability of GPUs by NVIDIA's CUDA and OpenCL (see Chapter 2.2.3). With the introduction of full double precision support on GPUs, many scientific projects started using e.g. finite difference and finite element techniques for the solution of systems of partial differential equations (PDEs) using GPU hardware. Since 2003, several papers described the solution of the Navier-Stokes equations for incompressible fluid flow on the GPUs [**BFGS03, KW03**] or other boundary value problems [**GLLS03**]. An analysis of a meteorological simulation for tropical cyclones and an implementation of rigid particle flows on GPUs can be found in [**HHR09b**].

With the introduction of built-in double-precision support and IEEE754 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 (i.e. a stream). Operations of a kernel are pipelined, such that many stream processors can process the stream in parallel. The limiting factor in this context is 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 other register sets can be done in just a few cycles, whereas the cost of fetching data from the global memory extends several hundreds of cycles.

While the problem described above is often inherent for many-core computing, other restrictions of stream processing techniques have been addressed in CUDA, which offer e.g. gather and scatter operations on the global graphics memory. Furthermore, CUDA-capable devices can be programmed with slightly extended C and runtime libraries, including hardware support for double precision (obeying IEEE 754).

In fall 2009 NVIDIA released their new chip architecture "Fermi". In the following, the utilization of this new architecture is for scientific computing is evaluated and compared to former generations.

5.2.1. Hardware Description. As hardware platform for the evaluation accelerators from the actual and previous generation have been chosen both from the NVIDIA professional line (Tesla), as well as from the NVIDIA consumer line (GeForce). A very detailed schematic overview of the GT200 respectively T10 GPU is given in Figure 2.4 in Chapter 2.1.4. In Table 5.2 the main architectural and theoretical performance characteristics of the GPUs are explained. Additionally, the systems hosting the GPUs are shown in Tabular 5.3 where also measurements of key system properties are given.

5.2.2. Elementary Kernels Performance Results. In a first step, measurements of the bandwidth on the host side of system as well as of the devices are given in Tabular 5.3. In a second step benchmarks of some elementary kernels are shown. Beside dot-products, vector updates, scalar-products, matrix-vector and matrix-matrix operations are performed both in single and double precision. All measured peak performances are also shown in Table 5.4. Figure 5.15 to 5.19 illustrate the progression of the performance curve of the respective kernel.

Name	Tesla C2050	Tesla C1060	GTX 480	GTX 280
Chip	T20	T10	GF100	GT200a
Transistors	ca. 3 Mrd.	ca. 1,4 Mrd.	ca. 3 Mrd.	ca. 1,4 Mrd.
Core frequency	1.15 GHz	1.3 GHz	1.4 GHz	1.3 GHz
Shaders (MADD)	448	240	480	240
GFLOPs (single)	1030	933	1.345	933
GFLOPs (double)	515	78	168	78
Memory	3 GB GDDR5	4 GB GDDR3	1.5 GB GDDR5	1 GB GDDR3
Memory Frequency	1.5 GHz	0.8 GHz	1.8 GHz	1.1 GHz
Memory Bandwidth	144 GB/s	102 GB/s	177 GB/s	141 GB/s
ECC Memory	yes	no	no	no
Power Consumption	247 W	187 W	250 W	236 W
IEEE double/single	yes/yes	yes/partial	yes/yes	yes/partial

Table 5.2. Key system characteristics of the four GPUs used for the tests. Computation rate and memory bandwidth are peak respectively theoretical values.

Host				Device				
CPU	MEM [GB]	BW [GB/s]	H2D [GB/s]	GPU	MEM [GB]	BW [GB/s]	D2H [GB/s]	CC ECC
2 x Intel Xeon (E5520, 4 cores)	32	12.07	PA: 3.25 PI: 5.86	Tesla T20	3	BT: 91.28 daxpy: 82.5 ddot: 88.3	PA: 2.51 PI: 4.75	2.0 Yes
2 x Intel Xeon (E5450, 4 cores)	16	6.14	PA: 1.92 PI: 5.44	Tesla T10	4	BT: 71.80 daxpy: 83.1 ddot: 83.3	PA: 1.55 PI: 3.77	1.3 No
1 x Intel Core2 (6600, 2 cores)	2	3.28	PA: 1.76 PI: 2.57	GTX 480	1.5	BT: 108.56 daxpy: 135.0 ddot: 146.7	PA: 1.38 PI: 1.82	2.0 No
1 x Intel Core i7 (920, 4 cores, SMT on)	6	12.07	PA:5.08 PI:5.64	GTX 280	1.0	BT: 111.54 daxpy: 124.3 ddot: 94.81	PA: 2.75 PI: 5.31	1.3 No

Table 5.3. Systems' configurations. The abbreviations are as follows: MEM is the amount of memory, BW the Bandwidth, H2D denotes the Host to Device bandwidth via PCIe and D2H the other way round, CC is the CUDA 'compute' capability and ECC is the availability of error correcting memory. PA means pageable memory is allocated, PI denotes the usage of pinned memory.

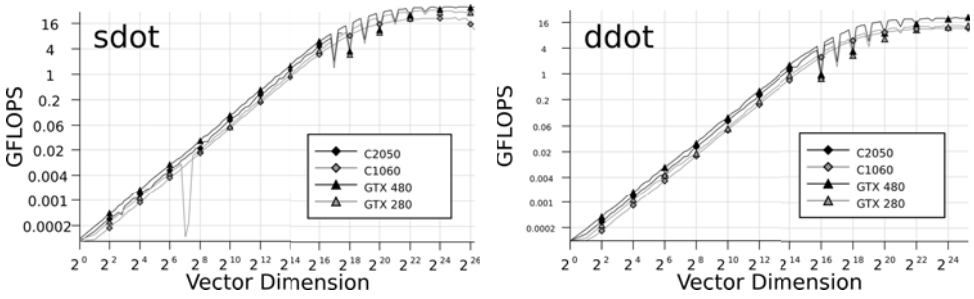


Figure 5.15. Performance of the dot routine performed in single (sdot) and double precision (ddot).

Except for the multiplication of two matrices with each other, all operations are limited by the bandwidth of the GPU memory and the performance stays significantly below the theoretical peak performance. Since there are not major changes in the architecture of the GPUs, the behavior is usually similar to each other.

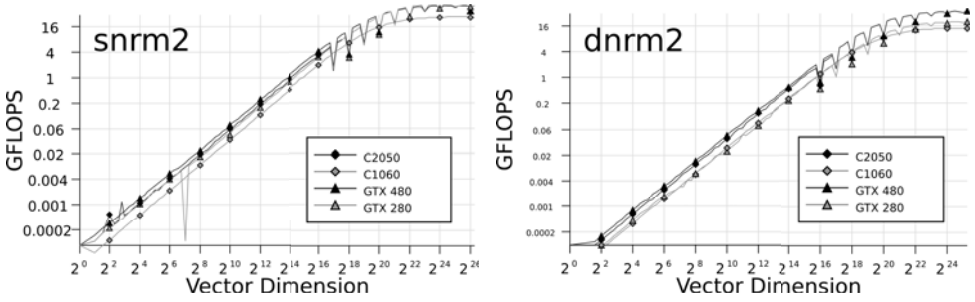


Figure 5.16. Performance of the 2-norm routine performed in single (snrm2) and double precision (dnrm2).

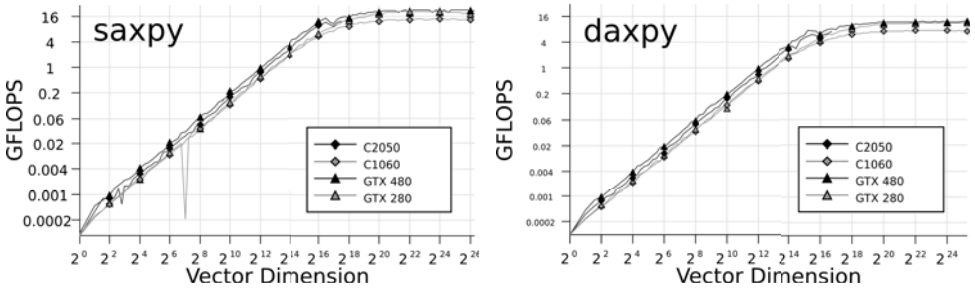


Figure 5.17. Performance of the vector scale and add routine axpy performed in single (saxpy) and double precision (daxpy).

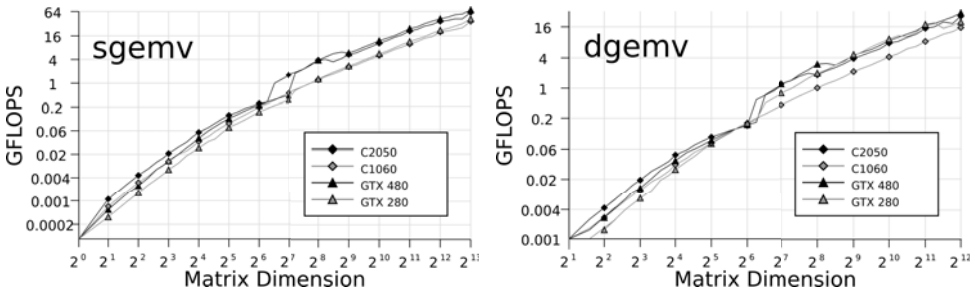


Figure 5.18. Performance of dense matrix vector multiplication routine performed in single (sgemv) and double precision (dgemv).

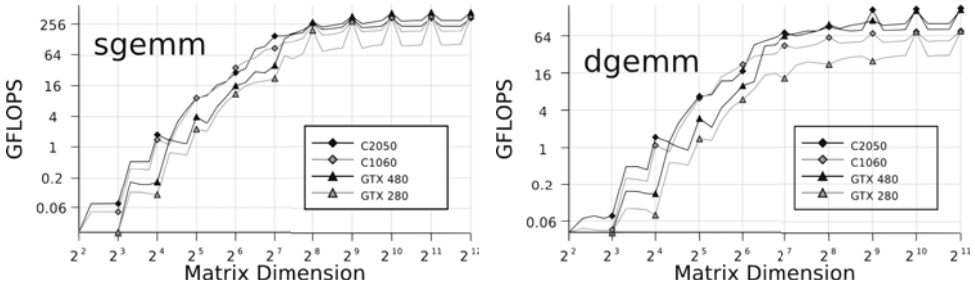


Figure 5.19. Performance of the general matrix matrix multiplication routine gemm performed in single (sgemm) and double precision (dgemm).

Experiment Setup		Performance (GFLOP/s)			
Routine	Data size	C2050	C1060	GTX 480	GTX 280
sdot	185364	9.27	6.50	12.36	8.24
sdot	39903170	29.83	18.41	38.17	29.11
ddot	110218	5.38	3.50	6.68	4.16
ddot	39903170	18.79	11.03	19.38	12.48
snrm2	185364	7.72	4.88	9.76	7.41
snrm2	39903170	44.34	26.47	48.72	48.99
dnrm2	110218	3.06	1.79	3.39	1.79
dnrm2	39903170	22,74	13.49	33.22	19.22
saxpy	185364	11.96	8.62	14.83	11.23
saxpy	39903170	23.83	13.23	22.08	19.16
daxpy	110218	6.68	4.59	7.87	5.80
daxpy	39903170	12.54	6,92	11.33	10.52
sgemv	8192	58.19	34.63	68.72	40.49
dgemv	4096	25.39	14.74	29.69	19.52
sgemm	4096	330.17	367.61	430.40	368.75
dgemm	2048	174.21	74.40	161.97	73.76

Table 5.4. Performance for the elementary kernels for special data sizes.

5.2.3. Iterative Refinement and Pure CG Solver. Evaluated is the performance of an implementation of the CG-algorithm (see Algorithm 7) based on the CSR-data format. The linear system is obtained from a finite element discretization of the Laplace equation on a unit square using linear test-functions, which is equivalent to a finite differences discretization based on the 5-point-stencil. The matrix has the following characteristics: 4.000.000 degrees of freedom (dofs) and 19.992.000 nonzero entries (nnz). All computations run exclusively on the accelerator and are performed in double precision, the absolute stopping criteria (see Chapter 1.6) for the residual is set to 10^{-6}.

First the performance results for the itemized kernels within the CG are presented and afterwards the complete runtime for the solver:

	C2050	C1060	GTX 480	GTX 280
ddot (sec)	0.000725	0.000768	0.000436	0.000675
ddot (Gbyte/s)	88.28	83.33	146.78	94.81
ddot (GFlops/s)	11.03	10.42	18.35	11.85
dscale+daxpy (sec)	0.00185	0.001916	0.001153	0.001285
dscale+daxpy (Gbyte/s)	51.89	50.10	83.26	74.71
dscale+daxpy (GFlop/s)	4.33	4.18	6.94	6.23
dcsrgemv (sec)	0.0187	0.019591	0.011527	0.013145
dcsrgemv (Gbyte/s)	17.10	16.33	27.75	24.34
dcsrgemv (GFlop/s)	2.14	2.04	3.47	3.04
daxpy (sec)	0.001163	0.001151	0.000711	0.000772
daxpy (Gbyte/s)	82.55	83.41	135.02	124.35
daxpy (GFlop/s)	6.88	6.95	11.25	10.36

Table 5.5. Performance evaluation of elementary kernels of the CG-algorithm on the four evaluated accelerators. All measurements are performed in double precision.

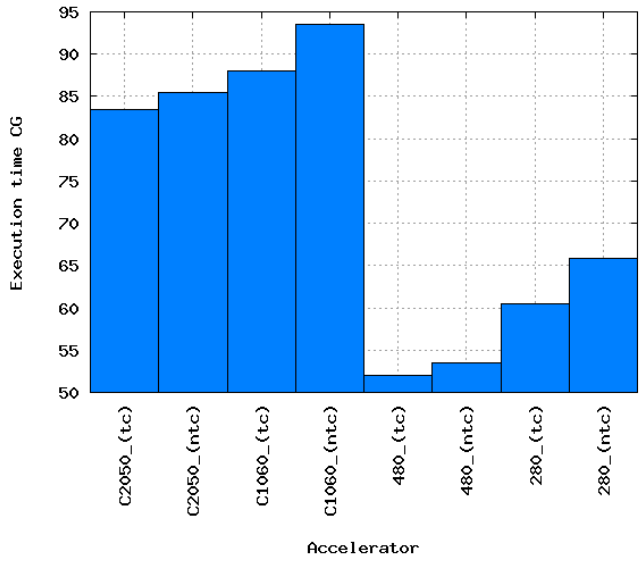


Figure 5.20. Runtime of the CG algorithm for the Laplace test case for the four evaluated accelerators. *tc* employs the use of texture cache, *ntc* without using it.

Beside the computational performance energy efficiency becomes more and more important for customers from academia and industry. Presented are results based on the theoretical peak power consumption of the accelerators from Tabular 5.2 and the runtime for the CG solver.

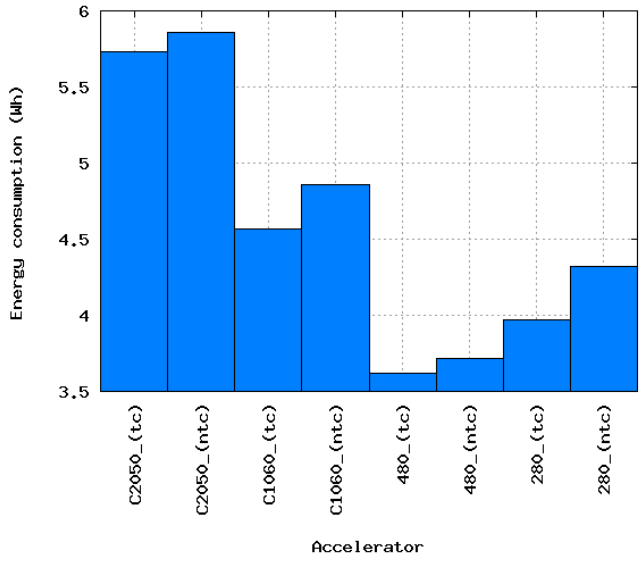


Figure 5.21. Energy consumption in Watt hours (Wh) for the Laplace test case for four evaluated accelerators. *tc* employs the texture cache, *ntc* does not.

To be able to evaluate the computational power of the hardware platform in a more complex application, a GPU-implementation of a plain GMRES-(30) solver is used and a mixed-precision iterative refinement implementation based on the same solver. Mixed precision iterative refinement methods, as described in Chapter 3.1

use a less complex floating point format for the inner error correction solver, and are therefore able to exploit the often superior low precision performance of GPUs.

Both, the plain double GMRES-(30) and the mixed precision variant use the relative residual stopping criterion of $\varepsilon = 10^{-10} \parallel r_0 \parallel_2$, while $\varepsilon_{\text{inner}} = 10^{-1} \parallel r_i \parallel_2$ was chosen as inner stopping criterion for the error correction variant. As right hand side a vector with ones for every entry and the zero-vector for the initial guess have been used.

In case of the here evaluated mixed precision iterative refinement implementation, the error correction solver is performed on the GPU, while the solution update is led to the CPU of the same system. This enables to handle larger problems, since the available memory on the GPU is usually very limited.

As test problems, three systems of linear equations CFD Venturi 2D 1, 2 and 3 affiliated with the 2D modeling of a Venturi Nozzle in different discretization fineness are used. Detailed information as well as sparsity plots for the smallest and largest CFD matrix can be found in Table 4.7. As base for the comparison, the total computation time has been chosen.

Experiment setup		Computation Time (s)					
problem	solver type	C2050	C1060	GTX 480	GTX 280	HC3	IC1
CFD 1	double	164.84	252.74	145.23	183.37	230.31	482.90
	mixed	80.48	129.19	60.98	98.46	91.71	195.59
CFD 2	double	473.38	778.75	456.17	518.49	819.46	1626.00
	mixed	273.99	510.38	256.43	301.41	401.57	896.94
CFD 3	double	993.63	1921.64	1145.08	1046.49	2493.33	4909.04
	mixed	554.28	1555.36	669.57	697.12	1330.70	2990.09

Table 5.6. Computation time ins s for problem CFD Venturi 2D 1, 2 and 3 based on a GMRES-(30). IC1 and HC3 results on 8 Cores

From Table 5.6 and Figure 5.22 it is visible that in terms of performance GPUs clearly outperform the CPUs from of the same generation (Intel Clovertown (IC1) vs. Nvidia T10/GT200 and Intel Nehalem (HC3) vs. Nvidia T20/GF100). This is at least true for the here evaluated problems. If the energy consumption of the standalone GPU is compared with the related value of a CPU the order stays manly the same.

But if the power consumption of the host of a GTX 480 is also taken into account it becomes clear that it has to consume less then 133 W to be as energy efficient as the HC3 (by assuming that the host performs no computation that reduces the runtime). Not considered is in this calculation the viewpoint of an computing center, where the power consumption without load and the performance per memory unit has also to be taken into account.

Performance

(1) GTX 480 (256 s)
(2) C2050 (273 s)
(3) GTX 280 (301 s)
(4) HC3 (401 s)
(5) C1060 (510 s)
(6) IC1 (896 s)

Energy Consumption

(1) GTX 480 (17,7 Wh)
(2) C2050 (18,7 Wh)
(3) GTX 280 (19,7 Wh)
(4) C1060 (26,5 Wh)
(5) HC3 (27,2 Wh)
(6) IC1 (127,93 Wh)

Figure 5.22. Performance and energy ranking for problem CFD Venturi 2D 2 based on a GMRES-(30). IC1 and HC3 results on 8 Cores. Energy efficiency for GPUs computed without energy consumption for the host.

5.3. FPGA-Accelerated Scientific Computing

FPGAs (Field-Programmable Gate Array) are integrated circuits and offer users the probability to modify the configuration after the manufacturing process. This is usually done by so called hardware description languages like VHDL or Verilog. In terms of complexity (number of transistors) and manufacturing method (usually defined in nm), FPGAs have the same characteristics like microprocessors but usually offer lower frequencies. In computational science FPGAs have a long tradition in the area of signal processing and related fields and some scientists assign them an important role in the medium-term future.

This chapter evaluates one representative of todays (2010) hybrid CPU-FPGA machines. First some elementary benchmarks are performed and after that numerical experiments based on the mixed precision iterative refinement technique for solving linear systems of equations are shown.

It has to be remarked that users programming FPGAs should be extremely cautious with respect to occurring rounding errors. This is especially the case when no IEEE rounding standards are available. Techniques like the iterative refinement method or interval arithmetics [**AH74**] could be considered in order to get a certain accuracy in IEEE respectively to obtain verified results.

5.3.1. Hardware Platform and Implementation Issues. All benchmarks have been performed on the Convey HC-1 which is described in more details in Chapter 2.1.5. Similar results to the presented ones can also be found in [**AHW10b**].

5.3.2. Elementary Kernels Performance Results. When comparing the results of the dot product and axpy operation (Figures 5.23 and 5.24), it is visible that between the single- and double-precision implementation, there is a factor of approximately two in terms of performance. There are almost no significant performance drops in relation to the length of the vector. For the *gemv* and *gemm* kernels there two effects can be observed. First, the double precision performance is always better then the single precision one. But for the *gemv* kernel it is much less then the factor of two. Second, the behavior of the performance depends strongly on the data size and the best results can be achieved when the matrices or vectors are a power of two. If such a value is exceeded, the performance breaks down.

To obtain the best performance on the Convey HC-1, it is very important to place the data in the right part of the memory. In Figure 5.27, the difference in performance for a axpy operation, performed in single precision, based on two available configurations for the allocation of memory is shown. Using the allocation function from the CML library, the performance breaks down, while a static allocation in the memory of the co-processor does not show such effects. The exact reason for this behavior has to be investigated in the future, but one presumption is, that the CML function does allocate the memory on the host side, and not on the co-processor part.

5.3.3. Linear Solver Benchmark Results. In the following, performance results of the CG solver and the iterative refinement method explained are shown. As test-case the Poisson equation on a unit-square has been chosen which has been discretized based on finite differences by means of a five point stencil leading to a linear system with 200 degrees of freedom. As right hand side a vector with ones for all entries has been chosen and the zero vector as initial solution.

The implementation of the CG solver has been done similar to Algorithm 7 where all calls to BLAS-operations have been replaced by calls to the Convey CML library.

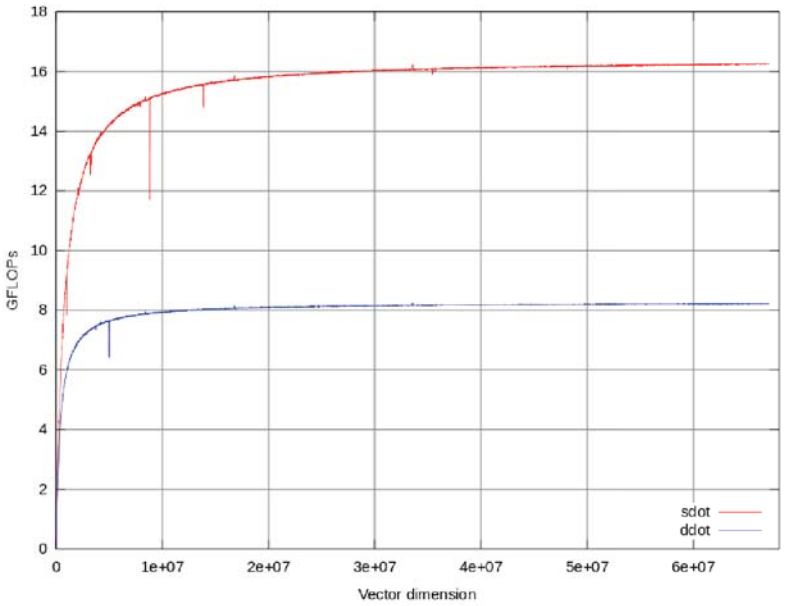


Figure 5.23. Performance results of the *dot-product* in single and double precision in GFLOPs as a function of the data size.

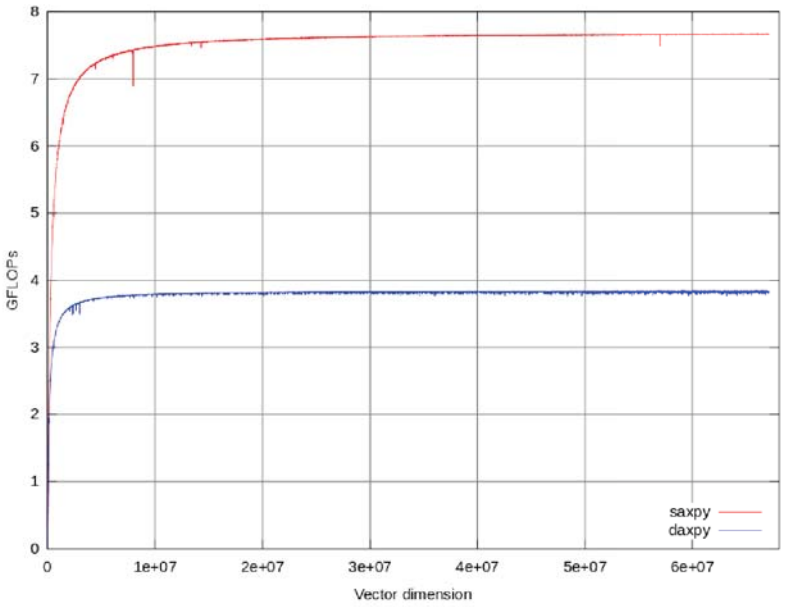


Figure 5.24. Performance results of the *axpy* operation in single and double precision in GFLOPs as a function of the data size.

Four configurations have been evaluated which differ in the two modes where the computation is performed and where the data is located. The first configuration is that all data is located in the host memory and the computation is also performed on the host. In the second configuration the data is placed in the co-processor memory but the computation is performed on the host (this is possible because there is one address space over the host and co-processor memory). In the third configuration the host as well as the co-processor performs computations and the associated memory holds the data. More precise, the matrix A and the vector p

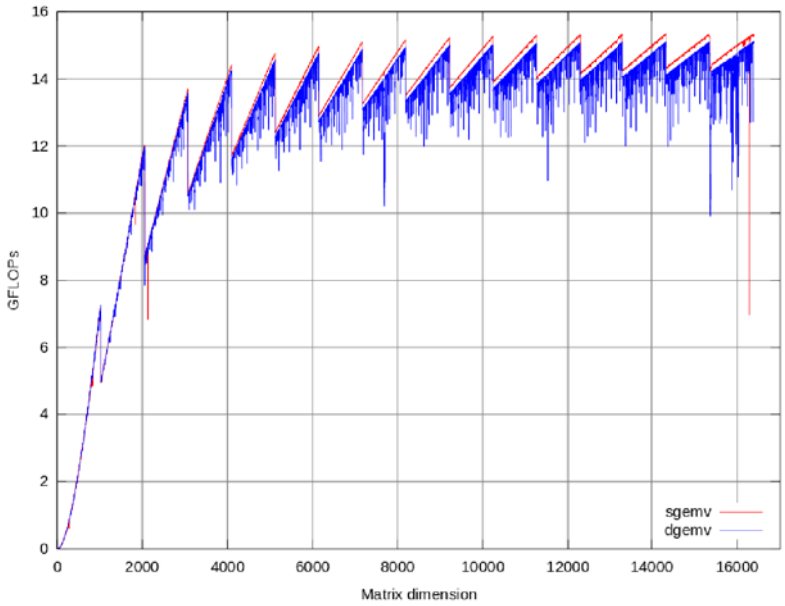


Figure 5.25. Performance results of the *gemv* operation in single and double precision in GFLOPs as a function of the data size.

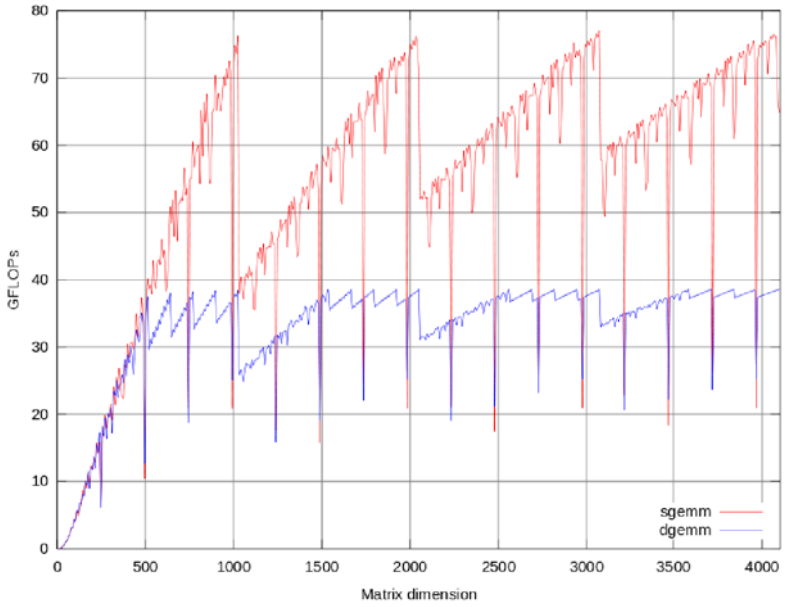


Figure 5.26. Performance results of the *gemm* operation in single and double precision in GFLOPs as a function of the matrix dimension.

is kept in the host memory, which is because the matrix vector multiplication is performed on the host, while all other vectors are kept in the device memory. Last, but not least all data is located in the host memory and the computations are all, except the mv-multiplication, executed on the co-processor.

Due to the fact that the CML does not support a matrix-vector multiplication in the CSR-format (see Chapter 4.1), this operation is always performed on the host.

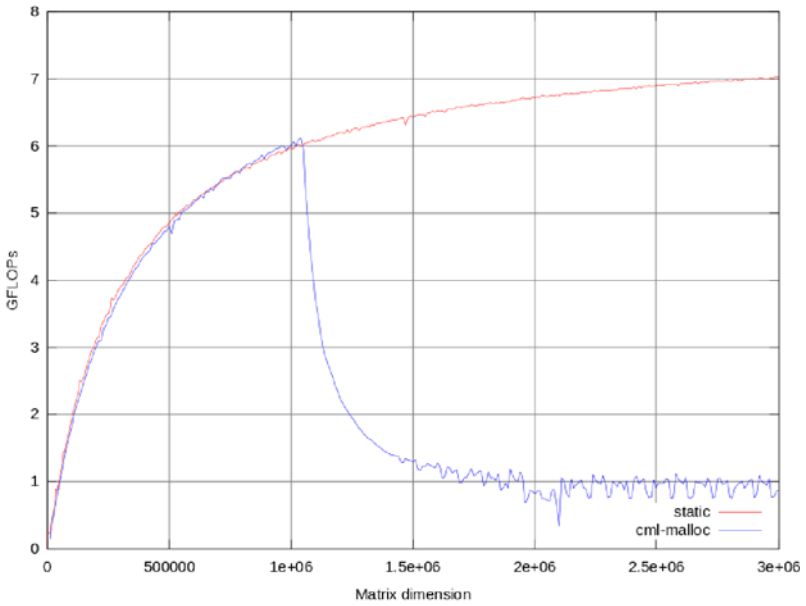


Figure 5.27. Evaluation of the saxpy performance using two different memory allocation procedures. When the cml-malloc is used the performance brakes down when a certain matrix size is reached.

Calculation	host	host	host,device	host,device
Memory	host	host,device	host,device	host
Total time	1.021	1.071	3.931	11.151
CG time	0.996	1.031	3.869	11.124
CSR mv time	0.717	0.726	2.653	0.728
BLAS time	0.229	0.232	1.115	10.353
OL time	0.022	0.034	0.056	0.024
CP calls	0	0	7180	7180

Table 5.7. Characteristics of the CG solver executed on the Convey HC-1 with different configurations. CP calls is the number of calls for a function that runs on the FPGA.

5.4. Inter-Architectural Comparison based on Iterative Refinement

5.4.1. Hardware Platform and Implementation Issues. For the performance evaluation and the consideration of energy efficiency, the choice was to take nodes of two HPC-clusters based on different generations of Xeon processors and one system accelerated by a Nvidia Tesla S1070. Both clusters used for the comparison are located at the Steinbuch Centre for Computing (SCC) at the Karlsruhe Institute for Technology (KIT).

The first machine is the Institutscluster IC1. As described in Chapter 5.1.1, it consists of five racks containing a total of 200 computing nodes, each equipped with two Intel quad-core Xeon 5355 processors with Clovertown architecture running at 2.667 GHz. There are 16 GB of main memory available on each node.

The second machine is a HP XC3000 cluster system, called HC3. In total, the cluster has 332 computation nodes, each equipped with two Intel quad-core Xeon

5540 with Nehalem architecture. 288 nodes own 24 GB of main memory, 32 own 48 GB and the last 12 are equipped with 128 GB.

The Tesla-based system consists of two nodes hosting two Intel Xeon 5450 processors operating at 3.0 GHz. A Tesla S1070 is connected via PCIe 2.0 16x and each node controls two Tesla T10 computing processors, both equipped with 4 GB of memory.

Due to the fact that in the presented experiments only the computing capabilities of one node is used and one T10 computing processor, the Tesla-system is in terms of performance equivalent to a system equipped with one Tesla C1060. For this reason, the energy consumption is calculated for the host alone and for one Tesla C1060.

	HC3	Tesla	IC1
Processors per node	2	2 CPUs / 1 GPU	2
Cores per processor	4	8 / 240	4
Theoretical comp. rate / core	10.1 GFlop/s	12 / 3.9 GFlop/s	10.7 GFlop/s
Theoretical comp. rate / node	81 GFlop/s	96 / 933 GFlop/s	85.3 GFlop/s
L2-cache per processor	8 MB	8 / - MB	8 MB
Nodes	278 / 32 / 12	-	200
Memory per node	24 / 48 / 144 GB	32 GB	16 GB
Memory full machine	10.3 TB	-	32 TB
Th. comp. rate full machine	27 TFlop/s	1.0 TFlop/s	17.6 TFlop/s
Power consumption load	80.8 kW	539 / 187,8 W[1]	103 kW

Table 5.8. Key system characteristics of the three machines used for the tests. The first five rows give an overview for one node followed by additional information concerning the full cluster systems HC3 and IC1.

5.4.2. Reference Examples. To be able to compare the performance of different implementation of the GMRES-(10) solver, tests with different linear systems are performed. In this work 10 denotes the restart parameter for the GMRES.

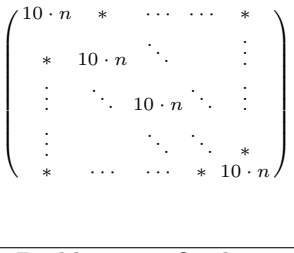
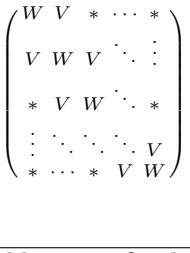
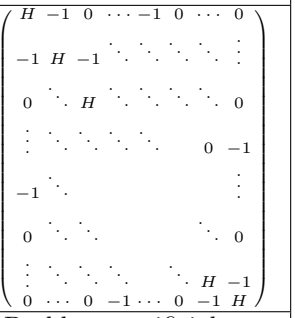
All solvers use the relative residual stopping criterion $\varepsilon = 10^{-10} \parallel r_0 \parallel_2$. Due to the iterative residual computation in the case of the plain GMRES-(10) solvers, the mixed GMRES-(10) solvers based on the mixed precision error correction method usually iterate to a better approximation since they compute the residual error explicitly, but as the difference is generally small, the solvers are comparable. In case of the mixed precision GMRES-(10) on the TESLA-System, the error correction solver is performed on one of the four available GPUs, while the solution update is led to the CPU of the same system. This is done to be able to handle larger problems since the amount of memory on the GPU is limited to 4 GB. The hardware platform is therefore similar to a system equipped with one TESLA C1060, but in the following the results are denoted with S1070.

On the one hand, matrices with a preset condition number are used, preset sparsities, and increase the dimension. Depending on the sparsity, the matrices are stored in the matrix array storage format (MAS) or the compressed row storage format (CRS).

M1 The first test matrix, is a dense matrix that is generated with the DLATMR-routine out of the Lapack library [**LAP**]. As parameter set, all entries are chosen smaller 1, and a condition number of 3. One drawback is that one cannot set positive definiteness in the routine itself. To ensure this property, the diagonal entries are set to be $10 \cdot n$, where n is the dimension of the matrix. By doing so, one loses control of the condition number, but it is bounded by the former choice.

M2 The second test case is a sparse matrix similar to a 5-point stencil. The difference is the the term $H = 4 + 10^{-3}$ instead of $H = 4$ on the diagonal. Furthermore, the second and fifth upper and lower diagonal is filled with -1. The term 10^{-3} on the diagonal is used to control the condition number.

M3 As third artificial test matrix an ill-conditioned dense matrix is chosen. As it is not easy to control the condition number of a dense matrix, chosen is the term $W = 2 \cdot 10^3 \cdot n + n$ on the diagonal. The term on the upper and lower second diagonal is $V = 10^3 \cdot n$, and the rest of the matrix is filled with random double precision numbers between 0 and 1. These entries are the only entries the cannot be controlled, and in one row, they can at most sum up to $(n-3) \cdot (1-\varepsilon)$, but they can also sum up to $(n-3) \cdot \varepsilon$, with $\varepsilon > 0$. Since the random numbers are for large dimension evenly distributed, it is assumed that they sum up to $0.5 \cdot (n-3)$.

M1	**M2**	**M3**
		
Problem: artificial Problem size: variable Sparsity: $nnz = n^2$ Cond. no.: < 3 Storage format: MAS	Problem: artificial Problem size: variable Sparsity: $nnz = n^2$ Cond. no.: ca. $8 \cdot 10^3$ Storage format: MAS	Problem: artificial Problem size: variable Sparsity: $nnz = 5n$ Cond. no.: ca. $8 \cdot 10^3$ Storage format: CRS

Tab. 1: Structure plots and properties of the artificial test-matrices

Beside the artificial test-cases, linear systems obtained from the area of CFD are taken into account. The three systems of linear equations CFD Venturi 2D 1, 2 and 3 are affiliated with the 2D modeling of a Venturi Nozzle in different discretization fineness. The distinct number of supporting points leads to different matrix characteristics concerning the dimension, the number of non-zeros, and the condition number. Below, the characteristics for the CFD Venturi 2D 3 matrix is shown, sparsity plots and information for the similar matrices 1 and 2 can be found in Tabular 4.7. CFD Venturi 2D 1 has about four hundred thousand degrees of freedom, CFD Venturi 2D 3 about one million.

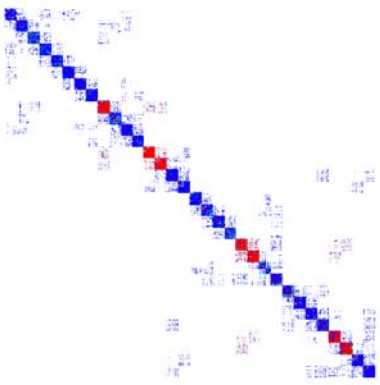
CFD Venturi 2D 2

Problem: CFD Venturi 2D 2, FEM Problem size: $n = 634.453$ Sparsity: $nnz = 5.700.633$ Symmetric: no Positive define: ?

Table 5.9. Sparsity plot and properties of the CFD Venturi 2D 2 test-matrix

5.4.3. Numerical Results. Figure 5.28, 5.29 and 5.30 show the results for the artificial test-cases for various dimensions on the GPU-based system as well as on the IC1. In Tabulars 5.10 to 5.12 show the results for the application based test-cases on all three machines.

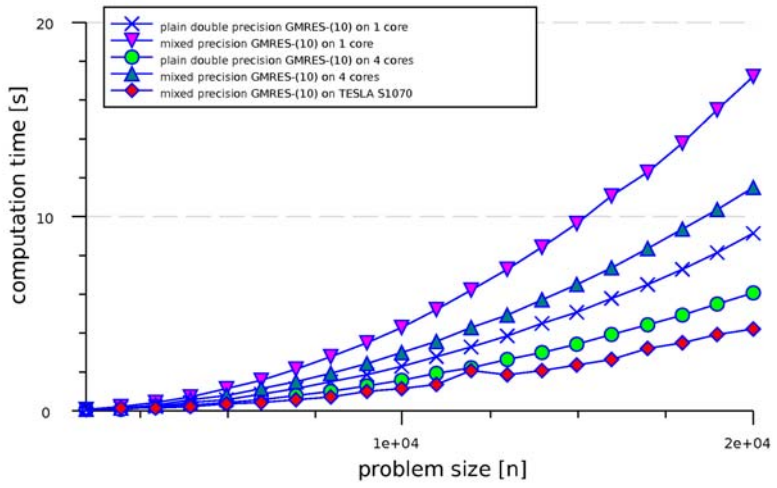


Figure 5.28. Test-case M1 executed on the TESLA-system and IC1. Relative residual stopping criterion $\varepsilon = 10^{-10}$;

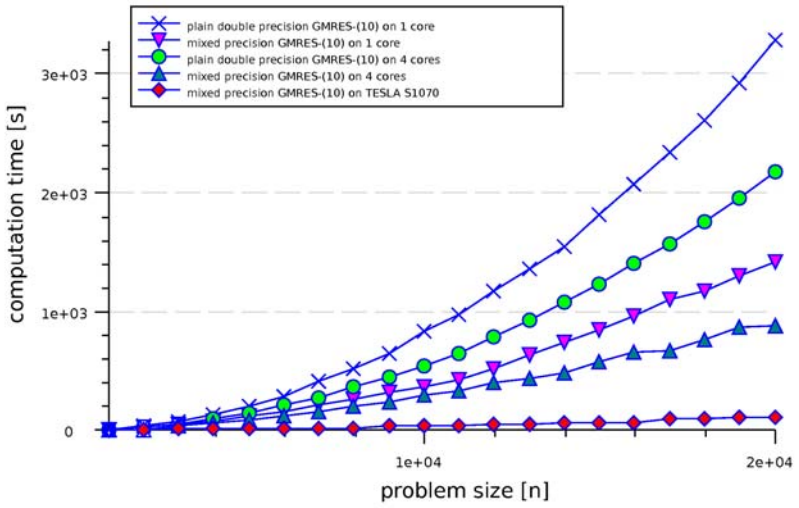


Figure 5.29. Test-case M2 executed on the TESLA-system and IC1. Relative residual stopping criterion $\varepsilon = 10^{-10}$;

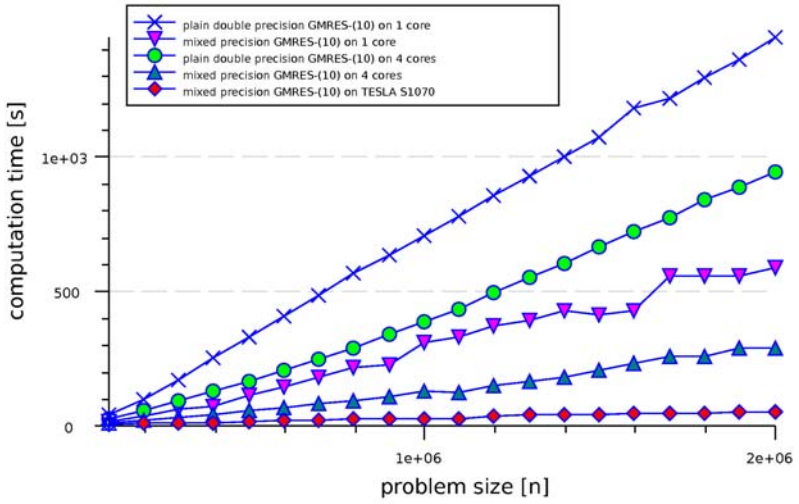


Figure 5.30. Test-case M3 executed on the TESLA-system and IC1. Relative residual stopping criterion $\varepsilon = 10^{-10}$;

		CPU-Cores			GPU
		1	4	8	
Computation time [s]	HC3 double	2267.47	1245.12	776.09	
	HC3 mixed	886.46	567.51	309.61	
	IC1 double	3146.61	1656.53	1627.77	
	IC1 mixed	1378.56	712.83	659.80	
	Tesla mixed				438.13

Table 5.10. Computation time for problem CFD 1 based on a GMRES-(10) as inner solver for the error correction method

		CPU-Cores			GPU
		1	4	8	
Computation time [s]	HC3 double	10765.30	4528.09	3363.44	
	HC3 mixed	4827.98	2177.19	1648.27	
	IC1 double	13204.70	6843.66	6673.07	
	IC1 mixed	5924.32	3495.09	3681.28	
	Tesla mixed				2092.84

Table 5.11. Computation time for problem CFD 2 based on a GMRES-(10) as inner solver for the error correction method

		CPU-Cores			GPU
		1	4	8	
Computation time [s]	HC3 double	62210.70	19954.50	16541.90	
	HC3 mixed	42919.80	9860.26	8828.28	
	IC1 double	60214.50	32875.10	32576.50	
	IC1 mixed	41927.40	19317.00	19836.80	
	Tesla mixed				10316.70

Table 5.12. Computation time for problem CFD 3 based on a GMRES-(10) as inner solver for the error correction method

5.4.4. Result Interpretation. In the first test (Figure 5.28), the low condition number leads to a good convergence rate of GMRES-(10) and after few iterations the solution approximation fulfills the stopping criterion. The additional computational cost of the mixed precision iterative refinement approach is large compared to the computational cost of the pure double solver. Therefore is the mixed precision GMRES-(10) neither for the sequential, nor for the parallel case able to compete with the plain double precision GMRES-(10). The TESLA S1070-implementation of the mixed GMRES-(10) outperforms the solvers on the IC1 due to the larger number of cores and the excellent single precision performance of the GPU, that can be exploited by the inner error correction solver. It should be mentioned, that the factors between the computation time of the different solver types are independent of the dimension n of the linear system that is solved.

The difference of the second test (Figure 5.29) to the first test case is the fairly high condition number of $\kappa \approx 8 \cdot 10^3$ of the linear system. Due to the high number of iterations the linear solvers have to perform, the overhead for the mixed precision method is considerably small. Therefore, also on the IC1, the additional costs can be overcompensated by the speedup gained by performing the inner solver in a lower precision format. Both, for the parallel and the sequential case a factor of about two is gained using the mixed precision iterative refinement approach instead of the plain double precision GMRES-(10). The lower precision format leads to a shorter execution time when performing elementary computations on the one hand, and to a more efficient use of the memory bandwidth on the other hand. The memory space needed to store one single precision floating point number is half the size that is needed for one double precision floating point number. The memory bandwidth is usually the limiting factor of the computational power of a system. Using a lower precision format, the processors have shorter waiting time for the data, and the system gains a higher efficiency. Since this argument applies to all memory levels, the speedup using single precision for the GMRES-(10) can even exceed the factor 2 that characterizes the speedup of a general purpose CPU when switching from double to single precision computations.

The speedup factor gained by performing the mixed GMRES-(10) on the TESLA S1070 is almost 15 with respect to the sequential plain double GMRES-(10) on the IC1. Again one can observe, that the speedup factors between the different solvers on the different Hardware platforms remain constant, independent of the problem size.

For the third test case (Figure 5.30), again an artificial test matrix is used with a condition number of $\kappa \approx 8 \cdot 10^3$. The difference to the former test cases is, that now the solvers are applied to sparse linear systems where the matrices are stored in the CRS format. The low number of nonzero entries leads no longer to a computational cost that is quadratically increasing with the dimension, but linearly. Furthermore is the total computational effort lower compared to the tests with matrix structure M2. Despite some perturbations, that can be explained by rounding effects and the use of different cache levels, one can still observe that the quotients between the solver types remain the same, independently of the dimension of the linear system. Again, both for the sequential and the parallel case, the mixed precision GMRES-(10) on the IC1 outperform the plain double implementations due to the fact, that the additional computational cost of the iterative refinement scheme is overcompensated by the speedup gained through the execution of the inner solver in single precision. The implementation on the TESLA S1070 can additionally exploit the excellent single precision performance of the highly parallelized GPU. Furthermore approximate the speedups gained by using the mixed precision GMRES-(10) the speedups of test case M2.

All tests with the matrices CFD1, CFD2 and CFD3 show that the mixed precision iterative refinement approach is also beneficial when applying solvers to real world problems. The mixed GMRES-(10) solvers outperform the plain double GMRES-(10) implementations for all test problems, both in the sequential and the parallel case. The reason is again the fact, that the additional computational cost of the iterative refinement approach is overcompensated by the cheaper inner solver using a lower precision format.

Using hybrid hardware, the mixed GMRES-(10) on the TESLA S1070 even generates speedups up to 7 with respect to the plain double implementation on the IC1. It can be observed, that this factor decreases for increasing dimension. The reason is, that for large data amounts, the connection between the host CPU and the GPU slows the mixed GMRES-(10) down.

5.4.5. Energy Efficiency. By using the values given in Table 5.8 for the power consumption P under load of the different architectures, 244 W per node can be obtained for the HC3, 514 W per node for the IC1 and 718 W for the Tesla-system (node plus energy for one Tesla C1060). With these values, it is possible to estimate the total amount of energy E that has been spent for a computation that has taken t seconds through the relation $E = P \cdot t$. The function indicates a linear characteristic due to the fact that one node is used, assuming a constant energy consumption, not taking into account whether one or more cores were used.

Modern processors usually offer the possibility of automatically raising the clock-speed if sequential code is executed, and deactivating parts of the processor, if they are not needed. The power consumption is effected by such mechanisms, and energy measurements have to be performed for every run. This becomes difficult when a machine is in production mode. On the HC3, measurements have shown an energy consumption of 243,5 W per node in case of performing complex computations using most components of the processor, and a power consumption of 225 W for less complex computations using only little resources. The system setting for the CPU-frequency is “on demand”. A deeper analyses of the machines may give more

detailed information, but is can be assumed, that the energy consumption will generally stay within these limits.

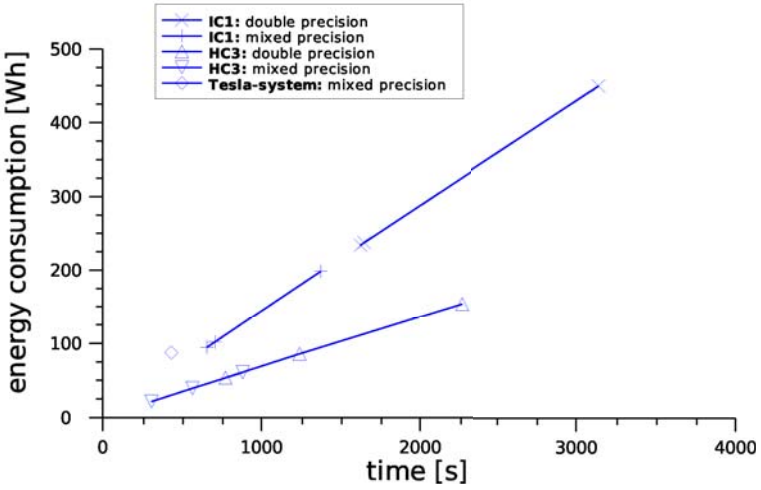


Figure 5.31. Energy consumption as a function of time for solving the CFD Venturi 2D 1 test-case on HC3, IC1 and Tesla. The inner solver is a GMRES-(10).

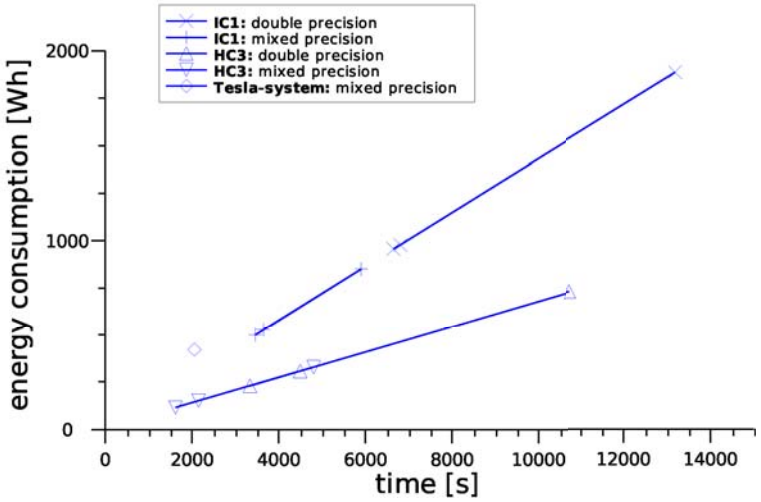


Figure 5.32. Energy consumption as a function of time for solving the CFD Venturi 2D 2 test-case on HC3, IC1 and Tesla. The inner solver is a GMRES-(10).

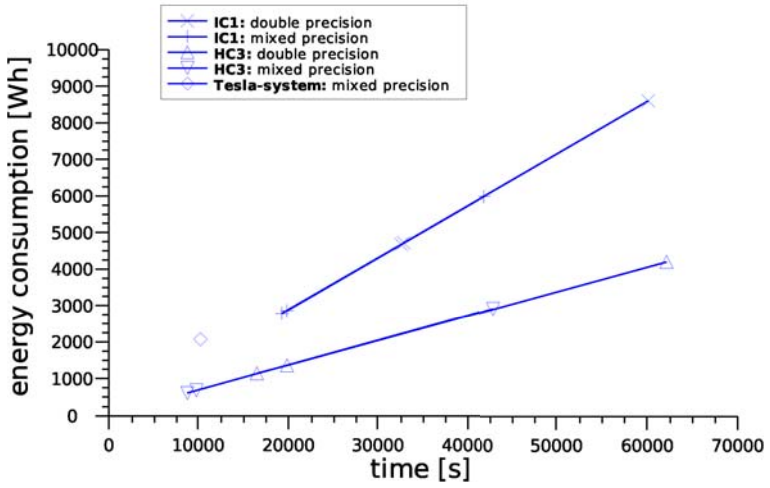


Figure 5.33. Energy consumption as a function of time for solving the CFD Venturi 2D 3 test-case on HC3, IC1 and Tesla. The inner solver is a GMRES-(10).

5.4.6. Conclusions. The numerical tests in the previous section have shown the high potential of using different precision formats within the proposed error correction solver. The obtained algorithm is flexible in terms of choosing the inner correction solver, and robust in terms of numerical stability. The possibility of performing the error correction solver on a coprocessor increases the potential of mixed precision methods, as they can be implemented efficiently on hybrid systems. Performing the error correction solver of an error correction method in a lower format leads to an overall increase in performance for a large number of problems.

On a CPU (IC1 and HC3), performing the error correction method in mixed precision, one often achieves a speedup factor of around two compared to the plain solver in double precision.

When using hybrid hardware, consisting of coprocessors specialized on low precision performance, even higher speedup factors can be expected. In the numerical experiments for the FEM discretizations of the Venturi Nozzle speedups of more than seven for the CUDA implementation are achieved.

Still, a very ill-conditioned problem can lead to a high number of additional outer iterations necessary to correct the rounding errors, that arise from the use of a lower precision format in the error correction solver. For the worst case, the inner solver will not converge. Due to the fact that one is usually not able to determine a priori whether the mixed precision method is superior for a specific problem, an optimized implementation of the solver would execute the first solution update of the mixed precision error correction method and determine, depending on the improvement of the solution approximation, whether it should continue in the mixed precision mode or whether it should use the plain solver in high precision. The next step beyond this strategy of changing between single and double precision is to use techniques around adaptive precision, where the precision is adjusted according to the convergence in the inner solver. FPGAs and related technologies may provide the capabilities for such algorithms.

For an efficient implementation of the mixed precision iterative refinement techniques in a solver suite, some additional work is required, especially concerning the use of preconditioners. This may not only increase the stability of the solver, but also its performance. In such an environment, the mixed precision error correction methods form powerful solvers for FEM simulations and beyond.

The iterative refinement method implemented with mixed precision techniques, and combined with GPU coprocessor technology shows very good results, at least for the presented test-cases arising in the field of computational fluid dynamics. This shows the high potential of hardware-aware computing. When numerical procedures are developed and implemented with respect to the available hardware resources, one can expect advantages in terms of performance and energy efficiency. Since, depending on the problem, the time and energy savings can become quite large, hardware-aware computing should be taken into account for individual solutions as well as for large-scale simulations in computing centers.

The Tesla-system is based on an old CPU-architecture, comparable to the IC1-cluster. All experiments have shown that even those older architectures can greatly benefit by adding accelerators like GPUs, both in terms of performance and energy efficiency. The GPU used in the Tesla-system is based on the old T10 computing chip, and with the availability of the new Fermi-based GPUs, even the newest CPU-architectures will be outperformed.

Future work includes a larger set of experiments concerning the linear systems, the used solvers, and the evaluated hardware platforms. One focus could also be to monitor the energy saving techniques of modern processors. Since the energy, the system needs in idle may be a point of interest as well, it could also be taken into account. The first step in this direction is already done in [**ACF$^+$11**].

CHAPTER 6

Impact of Data Distribution in Accuracy and Performance of Parallel Linear Algebra Subroutines

In parallel computing the data distribution may have a significant impact in the application performance and accuracy. These effects can be observed using the parallel matrix-vector multiplication routine from PBLAS with different grid configurations in data distribution. Matrix-vector multiplication is an especially important operation once it is widely used in numerical simulation (*e.g.*, iterative solvers for linear systems of equations as shown in the previous chapters).

This chapter presents a mathematical background of error propagation in elementary operations and proposes benchmarks to show how different grid configurations based on the two dimensional cyclic block distribution impacts accuracy and performance using parallel matrix-vector operations. The experimental results validate the theoretical findings.

Parts of the results of this chapter have been reviewed, presented and published in the context of the International Meeting on High Performance Computing for Computational Science (VECPAR 2010), organized by Lawrence Berkeley National Laboratory and University of Porto [**RKH11**].

6.1. Introduction

In many numerical algorithms, problems are reduced to a linear system of equations. Therefore, solving systems like $Ax = b$ with a matrix $A \in \mathbb{R}^{n \times n}$ and a right hand side $b \in \mathbb{R}^n$ is essential in numerical analysis. There are two major ways of solving those systems: by direct solvers, which are mainly based on the Gaussian algorithm, or by iterative solvers (see e.g. Chapter 1) which are often based on projections. The second type usually contains one multiplication of a matrix with a vector in each iteration step and the precision of such matrix-vector multiplication has a significant impact on the convergence of the iterative solver [**DDH07**].

In most modern microprocessors, mathematical operations are performed by using floating point arithmetics. However, the finite floating-point arithmetic can only deliver an approximation of the exact result due to rounding errors. Since the exact result is usually unknown, it is sometimes difficult to measure the quality of these approximations. Besides, as a result of several operations, the accumulation of those errors may have an impact in the accuracy of the results.

There are many papers proposing different solutions to find more accurate results. Some authors concern is to improve the numerical accuracy of the computed results in computers through the use of extra precise Iterative Refinement (see Chapter 3 or [**DHK$^+$06, DD05**]). Others try to use mixed-precision algorithms [**GHW06, LLL$^+$06, GST07, BBD$^+$08**] to obtain a good accuracy and improve the performance. Another possible way to deal with this unreliability is to

use verified computing [**HRKH97**]. Such techniques provide an interval result that surely contains the correct result [**AH74, AH83, KKL$^+$93, KM81**]. However, the use of such methods may increase the execution time significantly. This effect is even worse for large linear systems, that may need several days or even more to be solved. Based on these researches, it is possible to notice that there is a tradeoff performance versus accuracy.

Parallel computing is a well-known choice for simulating large problems. Since many numerical problems are solved via a large linear system of equations, a parallel algorithm would be a good approach. In this context, the libraries BLAS [**BDD$^+$02**] and LAPACK [**LAP**] seem a good choice, since they have a parallel version (PBLAS and SCALAPACK [**BCC$^+$96**]) that could be used in the case of very large systems. However, it is important to remember that these libraries provide an approximation of the correct result and not a verified result.

It is well known that the data distribution has a major impact in the performance of a parallel application [**BCC$^+$96**]. However, the data distribution can also present an important influence on the accuracy of the numerical results. Sometimes a fixed problem can lead to distinct solutions depending on the data distribution or the number of processes used in its solution. This effect can possibly be explained by the rounding error theory [**Lin70**].

Based on that, this work investigates the impact of different grids configuration used in the two-dimensional block cyclic distribution on the accuracy and performance of the parallel matrix-vector multiplication implemented by PBLAS. This particular distribution was chosen since it was proved to be a good choice for parallel matrix distribution on parallel environments with distributed memory [**DW93**]. Other interesting data distribution was proposed in [**EGPG96**], however it is also based on block distribution and was consider equivalent to the two-dimensional block cyclic distribution [**SH96**].

In this chapter, the performance of different grids configuration was measured and compared among them. To evaluate the accuracy of the approximations generated by PBLAS, a comparison with the verified solution provided by C-XSC [**KKL$^+$93**] is done. The experimental results indicate how the grids should be configured to find a compromise between accuracy and performance considering the application needs.

This chapter is organized as follows. To better understand this problem, Section 6.2 presents two important backgrounds: the theory of rounding errors and the two-dimensional block cyclic distribution scheme. Section 6.3 introduces the platform, input data and results obtained in the numerical experiments. Finally, Section 6.4 present some final remarks and considerations about future work.

6.2. Background

This section presents the background used in this Chapter to explain the observed effects of parallelization. Section 6.2.1 introduces the theoretical background from the mathematical point of view concerning rounding errors, based on a paper of Linz [**Lin70**]. Section 6.2.2 describes the two-dimensional block cyclic distribution used by PBLAS.

6.2.1. Theory of Rounding Errors. Let ϵ be the machine accuracy and $fl(a \circ b)$ the floating point result for an elementary composition of two real numbers a and b. An elementary operation $\circ \in \{+, -, *, /\}$ of a and b can be estimated with $fl(a \circ b) = (a \circ b) + \epsilon(a, b, \circ)$ for the worst case. We assume $A \in \mathbb{R}^{n \times n}, x, y \in \mathbb{R}^n$

and get $y_k = a_k x$ as result for the product $Ax = y$ for every entry $y_k \in y$. Let a_k denote the $k-th$ row of A. For all y_k, the approximation using the floating point arithmetic is $\hat{y}_k$.

The now explained approach for a summation is called "simple approach" or "simple strategy" in the following. For computing each $y_k \in y$ is to add the first entry to the next one and then add the following entries one by one to the previous result. Using floating point arithmetic and the abbreviation $fl(a_{k,i} \cdot x_i) = a_{k,i} \cdot x_i + \epsilon(a_{k,i}, x_i, \cdot) =: \hat{b}_{k,i}$, this strategy can be written as follows:

$$\hat{y}_{k_1} := (\hat{b}_{k,1} + \hat{b}_{k,2}) + \epsilon_1$$

$$\hat{y}_{k_i} := \hat{y}_{k_{i-1}} + \hat{b}_{k,i} + \epsilon_i = y_{k_i} + \sum_{j=1}^{i} \epsilon_j \ , i \in \{2, 3, \dots n-1\}$$

Let the representation be the normalized floating-point with binary exponent and q fraction bits and assume the addition to be done by truncating the exact sum to q bits. Let p_i be the exponent of $\hat{y}_{k_i}$ and $\nu = 2^{-q}$. The error for the ith step is then $|\epsilon_i| \leq \nu 2^{p_i}$ and the global error can be written using the estimates $a_{k,i} x_i \leq b$, $|\hat{y}_{k_i}| \leq ib$ and $2^{p_i} \leq 2ib$ in the following way

$$|y_k - \hat{y}_k| \leq \nu \sum_{i=1}^{n-1} 2^{p_i} \leq 2\nu b \sum_{i=1}^{n} i = \nu b n(n+1).$$

This means the error using this approach grows like $O(n^2)$.

A second strategy for the summation is the so called "Fan-In" algorithm, which we denote as "advanced approach". The values are added to each other in pairs and the algorithm is then executed recursively. Let us define the notation

$$y_k = \underbrace{\underbrace{\underbrace{a_{k,1}x_1 + a_{k,2}x_2}_{\hat{y}_{n_{1,1}}} + \underbrace{a_{k,3}x_3 + a_{k,4}x_4}_{\hat{y}_{n_{2,1}}}}_{\hat{y}_{n_{1,2}}} + \dots + a_{k,n}x_n}_{\hat{y}_{n_{1,m}}}$$

and

$$\hat{y}_{k_{i,j}} = \hat{y}_{k_{2i-1,j-1}} + \hat{y}_{k_{2i,j-1}} + \epsilon_{i,j}$$

where $\epsilon_{i,j}$ is the rounding error when computing $\hat{y}_{k_{i,j}}$. For the global error we have

$$|y_k - \hat{y}_k| = \sum_{\sigma_{1,k}} \epsilon_{i,j} \leq \nu \sum_{\sigma_{1,k}} 2^{p_{\sigma_{1,k}}}$$

where p is the exponent of the result and $\sigma_{1,k}$ is the set of all index pairs needed to get $\hat{y}_{k_{i,j}}$. Assuming that $a_{k,i} x_i \leq b$, we have:

$$\hat{y}_{k_{i,j}} \leq 2^j b \text{ and } 2^{p_{i,j}} \leq 2^{j+1} b.$$

Based on that, the error upper bound can be estimated by

$$|y_n - \hat{y}_k| \leq \nu \sum_{\sigma_{1,k}} 2^{p_{\sigma_{1,k}}} \leq 2\nu b \sum_{j=1}^{k} \sum_{i=1}^{n/2^j} 2^j = 2\nu b k n \leq 2\nu b \, n \log_2 n.$$

For the advanced approach the error grows like $O(n \log_2 n)$. The proof can be extended to cases in which more than two entries are added to each other using the "Fan-In"-algorithm. In that case, the error propagations is bounded by the one presented by the strategies above.

The two approaches differ in a factor of $n/(2 \log_2 n)$. This study suggests that a finer granularity in the summation leads to lower upper boundaries for rounding

errors. The proofs presented above show the impact of rounding errors in scalar products, which are commonly part of matrix-vector multiplications.

6.2.2. Data Distribution in Numerical Algorithms. On distributed memory platforms, the application programmer is responsible for assigning the data to each processor. How this is done has a major impact on the load balance and communication characteristics of the algorithm, and largely determines its performance and scalability [**BCC+96**].

PBLAS routines are implemented supposing the matrices are stored in the distributed memory according to the two-dimensional block cyclic distribution [**DW93**]. In this distribution, an M by N matrix is first decomposed into MB by NB blocks starting at its upper left corner. The distribution of a vector is done considering the vector as a column of the matrix. Suppose we have the following 10x10 matrix, a vector of length 10 an MB and NB equal 3. In this case, we would have the following blocks:

$$\left(\begin{array}{ccc|ccc|ccc|c} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} & A_{0,5} & A_{0,6} & A_{0,7} & A_{0,8} & A_{0,9} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & A_{1,5} & A_{1,6} & A_{1,7} & A_{1,8} & A_{1,9} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} & A_{2,5} & A_{2,6} & A_{2,7} & A_{2,8} & A_{2,9} \\ \hline A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & A_{3,5} & A_{3,6} & A_{3,7} & A_{3,8} & A_{3,9} \\ A_{4,0} & A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} & A_{4,5} & A_{4,6} & A_{4,7} & A_{4,8} & A_{4,9} \\ A_{5,0} & A_{5,1} & A_{5,2} & A_{5,3} & A_{5,4} & A_{5,5} & A_{5,6} & A_{5,7} & A_{5,8} & A_{5,9} \\ \hline A_{6,0} & A_{6,1} & A_{6,2} & A_{6,3} & A_{6,4} & A_{6,5} & A_{6,6} & A_{6,7} & A_{6,8} & A_{6,9} \\ A_{7,0} & A_{7,1} & A_{7,2} & A_{7,3} & A_{7,4} & A_{7,5} & A_{7,6} & A_{7,7} & A_{7,8} & A_{7,9} \\ A_{8,0} & A_{8,1} & A_{8,2} & A_{8,3} & A_{8,4} & A_{8,5} & A_{8,6} & A_{8,7} & A_{8,8} & A_{8,9} \\ \hline A_{9,0} & A_{9,1} & A_{9,2} & A_{9,3} & A_{9,4} & A_{9,5} & A_{9,6} & A_{9,7} & A_{9,8} & A_{9,9} \end{array}\right) \left(\begin{array}{c} b_0 \\ b_1 \\ b_2 \\ \hline b_3 \\ b_4 \\ b_5 \\ \hline b_6 \\ b_7 \\ b_8 \\ \hline b_9 \end{array}\right)$$

Suppose we have 4 processors. The process grid would be a 2x2 grid as follows:

$$\left(\begin{array}{c|c} P^0 & P^1 \\ \hline P^2 & P^3 \end{array}\right)$$

These blocks are then uniformly distributed across the process grid. Thus, every processor owns a collection of blocks [**BCC+96**]. The first row of blocks will be distributed among the first row of the processor grid, that means among P_0 and P_1, while the second row will be distributed among P_2 and P_3, and so on. For this example, we would have:

$$\left(\begin{array}{ccc|c} P^0 & P^1 & P^0 & P^1 \\ \hline P^2 & P^3 & P^2 & P^3 \\ \hline P^0 & P^1 & P^0 & P^1 \\ \hline P^2 & P^3 & P^2 & P^3 \end{array}\right) \left(\begin{array}{c} P^0 \\ \hline P^2 \\ \hline P^0 \\ \hline P^2 \end{array}\right)$$

According to this distribution, each processor would have the following data:

$$P^0: \left(\begin{array}{ccc|ccc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,6} & A_{0,7} & A_{0,8} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,6} & A_{1,7} & A_{1,8} \\ \underline{A_{2,0}} & \underline{A_{2,1}} & \underline{A_{2,2}} & \underline{A_{2,6}} & \underline{A_{2,7}} & \underline{A_{2,8}} \\ A_{6,0} & A_{6,1} & A_{6,2} & A_{6,6} & A_{6,7} & A_{6,8} \\ A_{7,0} & A_{7,1} & A_{7,2} & A_{7,6} & A_{7,7} & A_{7,8} \\ A_{8,0} & A_{8,1} & A_{8,2} & A_{8,6} & A_{8,7} & A_{8,8} \end{array}\right) \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_6 \\ b_7 \\ b_8 \end{pmatrix} \quad P^1: \left(\begin{array}{ccc|c} A_{0,3} & A_{0,4} & A_{0,5} & A_{0,9} \\ A_{1,3} & A_{1,4} & A_{1,5} & A_{1,9} \\ \underline{A_{2,3}} & \underline{A_{2,4}} & \underline{A_{2,5}} & \underline{A_{2,9}} \\ A_{6,3} & A_{6,4} & A_{6,5} & A_{6,9} \\ A_{7,3} & A_{7,4} & A_{7,5} & A_{7,9} \\ A_{8,3} & A_{8,4} & A_{8,5} & A_{8,9} \end{array}\right)$$

$$P^2: \left(\begin{array}{ccc|ccc} A_{3,0} & A_{3,1} & A_{3,2} & A_{3,6} & A_{3,7} & A_{3,8} \\ A_{4,0} & A_{4,1} & A_{4,2} & A_{4,6} & A_{4,7} & A_{4,8} \\ \underline{A_{5,0}} & \underline{A_{5,1}} & \underline{A_{5,2}} & \underline{A_{5,6}} & \underline{A_{5,7}} & \underline{A_{5,8}} \\ A_{9,0} & A_{9,1} & A_{9,2} & A_{9,6} & A_{9,7} & A_{9,8} \end{array}\right) \begin{pmatrix} b_3 \\ b_4 \\ b_5 \\ b_9 \end{pmatrix} \quad P^3: \left(\begin{array}{ccc|c} A_{3,3} & A_{3,4} & A_{3,5} & A_{3,9} \\ A_{4,3} & A_{4,4} & A_{4,5} & A_{4,9} \\ \underline{A_{5,3}} & \underline{A_{5,4}} & \underline{A_{5,5}} & \underline{A_{5,9}} \\ A_{9,3} & A_{9,4} & A_{9,5} & A_{9,9} \end{array}\right)$$

The two dimensional block cyclic distribution is usually not used when the matrix has a sparse data structure. Common storage formats, like CRS (compressed row storage), CCS (compressed column storage) etc., usually lead to distributions where a certain number of rows or columns are given to each process. The CRS presents a row-wise data distribution, that could be seen as a $np \times 1$ grid of processes in the two dimensional block cyclic distribution. In the same way, a $1 \times np$ grid could be interpreted as a column wise distribution, e.g. similar to a strategy taken when the CCS is used.

From the mathematical theory it is clear that the upper bound for the rounding errors occurring in a sparse matrix vector multiplication increases with the number of nonzero elements (nnz) per row. In the proof presented in the last subsection, the variable n should be replaced with nnz, assuming that the matrix data is sparse and the vector data has $O(n)$ entries.

It is important to mention that, for a matrix vector multiplication, no benefit in accuracy can be expected due to parallelization when this is done based on a row-wise data distribution and assuming that the sequential part on each process does not use techniques like "fan-in".

6.3. Numerical Experiments

This Section presents experimental results for different grid compositions. The accuracy and performance are the focus of these tests.

The results of the considered matrix-vector multiplication were computed using first the sequential BLAS-routines to obtain the sequential time. After this sequential test, we used the PBLAS-routines from the MKL package for different grid compositions. All results have been computed three times to avoid effects caused by hard- and software problems.

To evaluate which grid presents the best accuracy among the tested grids we analyzed the accuracy obtained by the parallel implementation through a comparison with a verified result. The library used to obtain the verified result was C-XSC, which stands for "**e**xtension for **s**cientific **c**omputing", and is a free programming tool for the development of numerical algorithms which provides highly accurate and automatically verified results. C-XSC does computations based on interval arithmetics and direct rounding, providing an enclosure of the exact solution, which is represented by an interval. This means that for a matrix-vector multiplication, C-XSC will deliver a vector of intervals, each entry of the vector containing an interval enclosure of the correct solution. The diameter of the intervals is usually very small, since C-XSC implementation uses techniques to iteratively reduce the interval diameter proofing that the interval result includes the exact result [**KKL+93**].

The average error from the PBLAS result to the C-XSC result is computed as follows. First, for each component of the result vector it is checked if it is in the interior of the interval given from C-XSC. If it is inside the interval, it is considered correct. If it is not inside, the distance to the interval is stored. Then the arithmetic mean over all distances is computed, which we denote as average error.

Next Section introduces the platform used for the experiments. Section 6.3.2 illustrates the input data of four different matrices used in the tests. Finally Section 6.3.3 presents the accuracy and performance results with some considerations.

6.3.1. Hardware- and Software-Platform. The software platform used for executing the numerical experiments is composed of optimized versions of the library PBLAS (Intel CMKL in version 10.0.2.018 for test case M1 and version 10.1.2.024 for the test cases M2, M3 and M4), C-XSC version 2.2.3 and the standard Message Passing Interface (MPI, see Chapter 2.2.2), more precise the OpenMPI implementation in version 1.2.8. The compiler used in these tests was the Intel compiler in version 10.1.021.

As hardware platform the Institutscluster located at the Steinbuch Computing Centre (SCC) at the Karlsruhe Institute of Technology (KIT) was chosen. It consists of 200 computing nodes each equipped with two Intel quadcore EM64T Xeon 5355 processors running at 2,667 GHZ, 16 GB of main memory and an Infiniband 4x DDR interconnect. 17,57 TFlops is the overall peak performance of the whole system and 15,2 TFlops in the Linpack benchmark. One process was placed on one node and the allocated resources for the experiments were allocated exclusively to avoid effects from other programs. All results have been computed at least five times to get reliable results.

6.3.2. Input Data. All results shown in Chapter 6.3.3 refer to four different input matrices and vectors. Their properties can be seen in Tables 6.1 and 6.2.

M1, the first test matrix, has dimension 16000 and is filled with pseudo random numbers from the interval $[-0,5; 0.5]$.

M2 is a square matrix with dimension 4929 used as the initial basis for constrained nonlinear optimization problem represented by GEMAT1 which is the Jacobian matrix for an approximately 2400 bus system in the Western United States. M2, arising in the area of power flow, is a sparse matrix with a medium condition number, as presented in Table 6.1. It is important to mention that no special data storage is used for sparse matrices. They are always stored as dense matrices.

The third matrix tested was M3. As shows Table 6.2, it is a dense matrix with dimension 66. This matrix is used in the generalized eigenvalue problem $Kx = \lambda Mx$, where M3 is matrix K and matrix M is BCSSTM02, from BCSSTRUC1 set. This matrix arises in dynamic analysis in structural engineering.

Table 6.2 also presents the properties of matrix M4. It is a sparse matrix with dimension 765 which presents a very high condition number. This matrix is used in the real application of nonlinear radiative transfer and statistical equilibrium in astrophysics.

All vectors used for the experiments are filled with pseudo random numbers from the interval $[-0,5; 0.5]$.

Detailed information about the creation and properties of the test cases M2 to M4 can be found on the Matrix Market website [**Mar**]. In the following Chapter 7, two more matrices are considered for a similar evaluation.

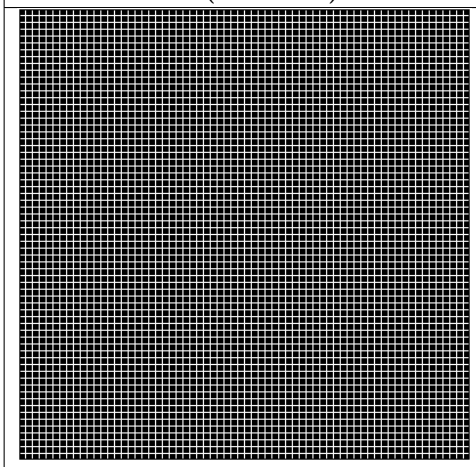
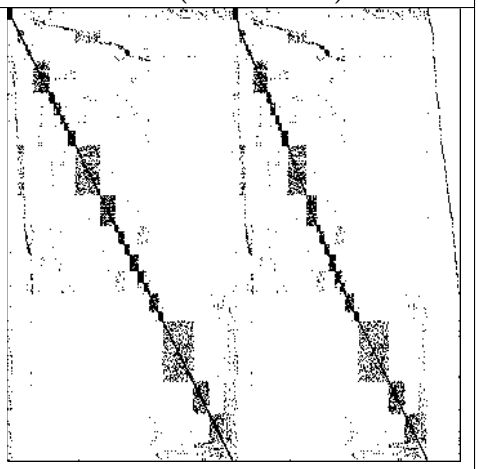
M1 (Random)	**M2 (GEMAT11)**
	
problem: Artificial problem size: $n = 16000$ sparsity: $nnz = 256000000$ cond. number: n.a. Frobenius norm: n.a.	problem: Power flow. problem size: $n = 4929$ sparsity: $nnz = 33185$ cond. number: 3.74e+08 Frobenius norm: 8.2e+02

Table 6.1. Sparsity plots and properties of the test-matrices M1 and M2

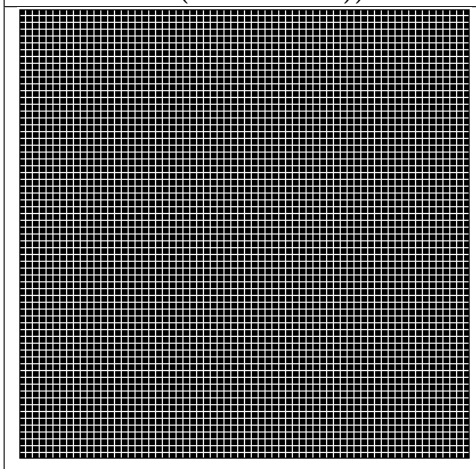
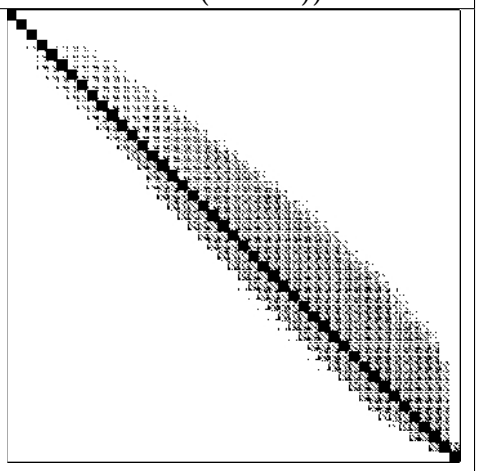
M3 (BCSSTK02))	**M4 (MCFE))**
	
problem: Structural eng. problem size: $n = 66$ sparsity: $nnz = 2211$ cond. number: 1.3e+04 Frobenius norm: 5.3e+04	problem: Astrophysics problem size: $n = 765$ sparsity: $nnz = 24382$ cond. number: 1.7e+14 Frobenius norm: 2e+17

Table 6.2. Sparsity plots and properties of the test-matrices M3 and M4

6.3.3. Numerical Results. This section discusses the results of a set of experiments using the four different matrices presented in the previous section. The first analysis is based on the accuracy obtained using different grid sizes. After that, the performance results are shown.

Figure 6.1 presents the accuracy obtained using nine processes and different grid configurations for test cases M1 and M2. The comparison between our PBLAS algorithm and the verified results of the C-XSC-algorithm show that there are constellations of processors grids, namely when the grid is $np \times 1$, where the accuracy of the parallel computation is as precise as the sequential one. In all other cases when the grid of processors is not $np \times 1$, the results present a better accuracy, suggesting that the accuracy depends on the grid. Another important observation is that the optimal accuracy for a fixed number of processes can be found by a $1 \times np$ grid. In general we observed that the more columns the processes grid have, the better is the accuracy.

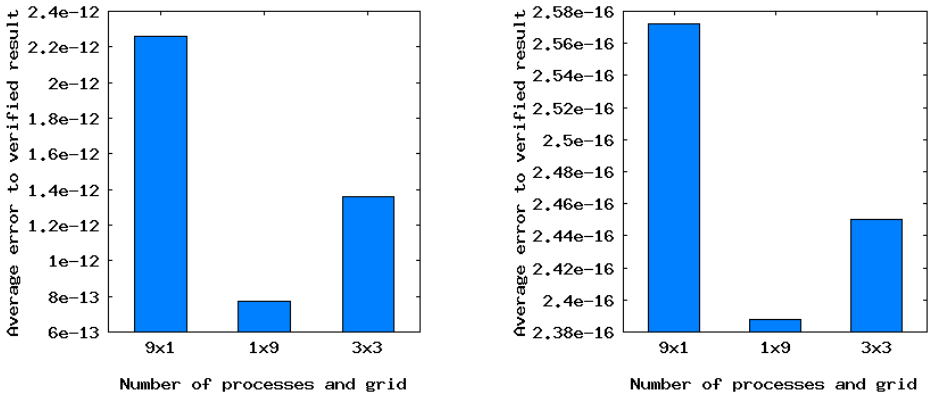


Figure 6.1. Average error to the verified result for three different grids of processes and a fixed number of nine processes for the test cases M1 (left plot) and M2 (right plot).

Investigating the example in the light of Chapter 6.2.1 using, for simplicity, a number of four processes. Let the dimension of the matrix be n, the number of processes $np = 4$, the grid of the processes $nr \times nc$ and the block size $nb := \frac{n}{np}$. Considering the case of an 2×2 grid of processes, the distribution follows the scheme in the example in Section 6.2.2. For a 1×4 and 4×1 grid, the notation P^a_{Bc} is used, where a (from 1 to np) represents the number of the processes containing the data, B denotes if it is a matrix(M) or a vector(v) and c is the number of the data block related to one processor. The data distribution is as presented in Tables 6.3(a) and (b).

Analyzing Table 6.4 and (b), it is possible to notice that in the computation based on a $np \times 1$ grid, each entry of the result vector is computed in just one process, which means that the summation is done like in the simple approach from Chapter 6.2.1.

$$\left(\begin{array}{c|c|c|c} P^0_{M0} & P^1_{M0} & P^2_{M0} & P^3_{M0} \\ \hline P^0_{M1} & P^1_{M1} & P^2_{M1} & P^3_{M1} \\ \hline P^0_{M2} & P^1_{M2} & P^2_{M2} & P^3_{M2} \\ \hline P^0_{M3} & P^1_{M3} & P^2_{M3} & P^3_{M3} \end{array}\right)\left(\begin{array}{c} P^0_{v0} \\ \hline P^0_{v1} \\ \hline P^0_{v2} \\ \hline P^0_{v3} \end{array}\right)$$

(a) Grid (1×4)

$$\left(\begin{array}{c|c|c|c} P^0_{M0} & P^0_{M1} & P^0_{M2} & P^0_{M3} \\ \hline P^1_{M0} & P^1_{M1} & P^1_{M2} & P^1_{M3} \\ \hline P^2_{M0} & P^2_{M1} & P^2_{M2} & P^2_{M3} \\ \hline P^3_{M0} & P^3_{M1} & P^3_{M2} & P^3_{M3} \end{array}\right)\left(\begin{array}{c} P^0_{v0} \\ \hline P^1_{v0} \\ \hline P^2_{v0} \\ \hline P^3_{v0} \end{array}\right)$$

(b) Grid (4×1)

Table 6.3. Data distribution for two different grids of four processes

The structure of the result distribution is shown in Table 6.4. The leading entry is the position of the resulting vector, followed by the explanation of which parts were combined to compute the result.

$$\left(\begin{array}{c} P^0(P^0_{M0}P^0_{v0} + P^1_{M0}P^0_{v1} + P^2_{M0}P^0_{v2} + P^3_{M0}P^0_{v3}) \\ \hline P^0(P^0_{M1}P^0_{v0} + P^1_{M1}P^0_{v1} + P^2_{M1}P^0_{v2} + P^3_{M1}P^0_{v3}) \\ \hline P^0(P^0_{M2}P^0_{v0} + P^1_{M2}P^0_{v1} + P^2_{M2}P^0_{v2} + P^3_{M2}P^0_{v3}) \\ \hline P^0(P^0_{M3}P^0_{v0} + P^1_{M3}P^0_{v1} + P^2_{M3}P^0_{v2} + P^3_{M3}P^0_{v3}) \end{array}\right)$$

(a) Grid (1×4)

$$\left(\begin{array}{c} P^0(P^0_{M0}P^0_{v0} + P^0_{M1}P^1_{v0} + P^0_{M2}P^2_{v0} + P^0_{M3}P^3_{v0}) \\ \hline P^1(P^1_{M1}P^0_{v0} + P^1_{M1}P^1_{v0} + P^1_{M1}P^2_{v0} + P^1_{M1}P^3_{v0}) \\ \hline P^2(P^2_{M2}P^0_{v0} + P^2_{M2}P^1_{v0} + P^2_{M2}P^2_{v0} + P^2_{M2}P^3_{v0}) \\ \hline P^3(P^3_{M0}P^0_{v0} + P^3_{M1}P^1_{v0} + P^3_{M2}P^2_{v0} + P^3_{M3}P^3_{v0}) \end{array}\right)$$

(b) Grid (4×1)

Table 6.4. Processes which contain the final result and parts from which it is computed

The advanced approach presented in Chapter 6.2.1 can be found on the $1 \times np$ grid where the final result will be placed on process one, but there are intermediate results on every process leading to a higher quality in the computed result. This suggests that the number of columns of the processor grid is responsible for the granularity of the computation - a higher numbers of columns can lead to better accuracy. So it is not astonishing that a symmetric grid produces results with an intermediate precision (bounded by the other grids).

It is also possible to notice that the results for the application based problem M2 show that for all grid sizes the average error is less than $2.58e-16$ which is excellent considering the double precision format.

Figure 6.2 presents the average error for matrix M3 and M4. Based on the M3 graphic, we can see that even for small problems, the data distribution and the executed computations can have an impact on the result. The average error to the verified result, depending on the grid configuration, differs in about one magnitude.

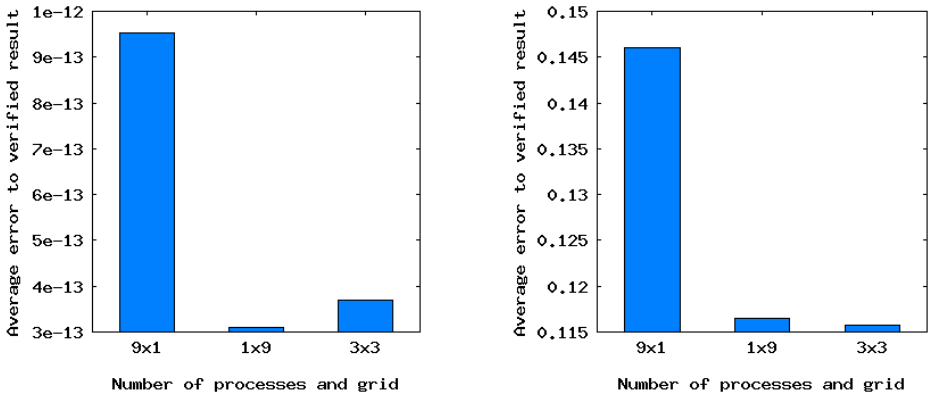


Figure 6.2. Average error to the verified result for three different grids of processes and a fixed number of nine processes for the test cases M3 (left plot) and M4 (right plot).

For the test case M4 we observe that the 9×1 grid delivers, analogue to all other experiments, the most inaccurate result. The fact that the 3×3 grid is a little more accurate than the 1×9 grid might be astonishing on the first view but this is possible because the mathematical theory gives only a upper bound for the rounding error propagation.

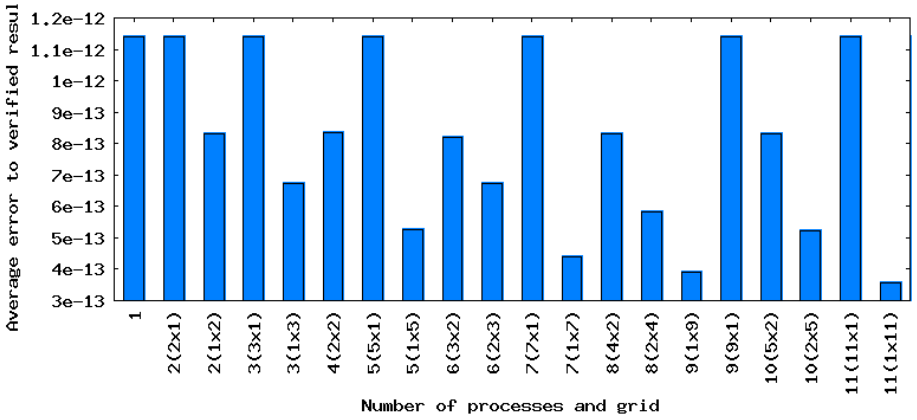


Figure 6.3. Average error to the verified result for different grids and different numbers of processes. A matrix similar to M1 but with dimension 8192 was taken for the experiments.

The results in Figure 6.3 show for different number of processes and grids of processes the average error to the verified result based on a matrix with dimension 8192 and input data generated like in M1. For all grids $np \times 1$ the accuracy is like in the sequential case and independent of the number of processes. The graphic shows that the more processes are used the better is the result. It can also be observed that the larger the number of processes, the better is the accuracy, following a logarithmic behavior, which corresponds to the theoretical findings.

6.3.3.1. *Performance.* This section presents the performance analysis considering matrix M1. The performance analysis for matrices 2 to 4 were not discussed since they have small dimensions. In this case it is not worth to parallelize the multiplication, since the program would spend more time communicating among the

processors than computing the result. Therefore it would maybe increase the computational time instead of speedup the computation. Since matrix 1 has dimension 16000, it is the natural choice for the performance test.

Figure 6.4 shows that directly interrelated to the grid is the processing speed. It is possible to notice that the computational time for the same problem size is very different depending on the grid.

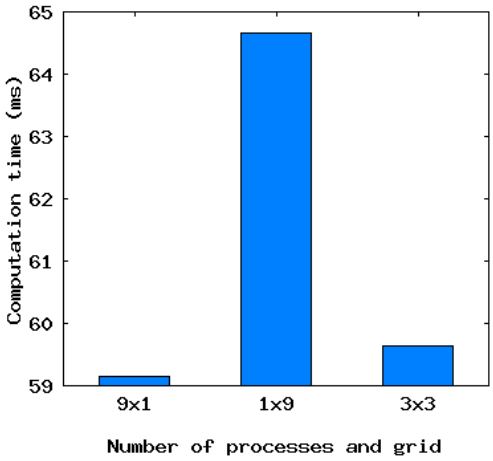


Figure 6.4. Commutation time for three different grids of processes and a fixed number of nine processes for the test case M1.

This performance variation can be explained by the fact that different grids communicate differently. The amount, length and topology of such communication have a significant impact on the performance [**DW93**]. In Table 6.3 is possible to notice that some of the data that a processes need may be stored in another processes, and therefore the processes need to communicate before the computation to send/receive parts these data. This occurs also after the computation, when parts of the result have to be collected from each processor and accumulated so that the result is found. This is the case of the 1×4 grid in Table 6.4.

The communication load-balance is optimal if it is equally distributed on all processes. The impact on the performance depends significantly on the underlying hardware (interconnect, memory bandwidth etc.). This means that not a single process is sending or receiving a big bunch of data to all other processes, but that all processes are sending little bunches of data to all other processes. For the grids shown above, the structure of the communication is:

(a) Grid (1×4)

before computation		*after computation*	
sender $\rightarrow$ *receiver*	*size*	*sender* $\rightarrow$ *receiver*	*size*
$P^0 \rightarrow P^1$	nb	$P^1 \rightarrow P^0$	n
$P^0 \rightarrow P^2$	nb	$P^2 \rightarrow P^0$	n
$P^0 \rightarrow P^3$	nb	$P^3 \rightarrow P^0$	n

(b) Grid (4×1)

before computation		*after computation*	
sender $\rightarrow$ *receiver*	*size*	*sender* $\rightarrow$ *receiver*	*size*
$P^0 \rightarrow P^1$	nb	–	–
$P^0 \rightarrow P^2$	nb	–	–
$P^0 \rightarrow P^3$	nb	–	–
$P^1 \rightarrow P^0$	nb	–	–
$P^1 \rightarrow P^2$	nb	–	–
$P^1 \rightarrow P^3$	nb	–	–
$P^2 \rightarrow P^0$	nb	–	–
$P^2 \rightarrow P^1$	nb	–	–
$P^2 \rightarrow P^3$	nb	–	–
$P^3 \rightarrow P^0$	nb	–	–
$P^3 \rightarrow P^1$	nb	–	–
$P^3 \rightarrow P^2$	nb	–	–

(c) Grid (2×2)

before computation		*after computation*	
sender $\rightarrow$ *receiver*	*size*	*sender* $\rightarrow$ *receiver*	*size*
$P^0 \rightarrow P^2$	$2 * nb$	$P^1 \rightarrow P^0$	$n/2$
$P^2 \rightarrow P^1$	$2 * nb$	$P^3 \rightarrow P^2$	$n/2$
$P^2 \rightarrow P^3$	$2 * nb$		

6.4. Remarks and Prospects

This Chapter presents the theory of rounding error propagation for elementary kernels and validates the theoretical findings based on numerical experiments. Beside accuracy the influence on performance of the process-grid using the two-dimensional block cyclic distribution was addressed. Tests show that the process-grid has a significant impact on both, but in a different way. The experiments suggested that the more columns the grid has, the better is the accuracy. However, this is not true for the performance, in which the effect is the opposite: the more columns the grid has, the worse is the performance. For symmetric grids, the performance achieved was good due to a better balance in the communication process. It presented, however, a little less accuracy then in the best case.

It is important to mention, that the impact of the data distribution on the results of numerical simulations depend strongly on the particular problem as well as on the numerical procedures employed to find the solution. Results showing the impact of the granularity of the parallelization can also be found in Table 4.15, where the number of iterations CG needs to solve the problem decreases when the resources are increased. The reason for this behavior can be explained by the above shown theory.

Ongoing research is the evaluation of different hardware platforms like CPUs from various vendors, GPUs and other accelerators as well as full cluster systems based on different interconnects. A second focus is the impact of the utilized software like compilers, optimization flags and different libraries. One goal is to accelerate linear solvers like CG or GMRES by performing a data reordering during the solution process. Within the next chapter, experiments are shown for the impact of the data distribution on a Jacobi-preconditioned CG-solver for two test-cases.

CHAPTER 7

Impact of Data Distributions in Accuracy on Krylov Subspace Methods

7.1. Introduction

In the previous chapter it is shown that the data distribution and related execution scheme of parallel executed basic linear algebra subroutines can influence accuracy as well as performance. The two dimensional block-cyclic data distribution was chosen since it is a quasi-standard used in many implementations like ScaLAPACK or PBLAS. Depending on the order the data is computed, rounding errors can occur in different patterns and propagate in different ways.

Error propagation is well investigated for direct linear solvers and splitting methods (see [**Hig96**]) but only few results can be found for projection methods, especially when they are used in combination with preconditioners. In practice however, the popular Krylov-subspace solvers CG and GMRES (see Chapter 1.3 and 1.4) are commonly used in combination with an adequate preconditioning (see Chapter 1.5 for basic preconditioners and Chapter 4.9 for their influence on the convergence of a solver) depending on the characteristics of the application. For some combinations of solver and preconditioner it is proofed that convergence can not be guaranteed. In general, the problem of influencing the convergence in numerical experiments becomes even more complex since rounding errors have also be taken into account. Beside numerical analysis, experiments have to be performed for solving problems in an efficient way.

The goal of this chapter is to evaluate the impact of different data distributions on the convergence of a parallel CG solver in combination with a Jacobi preconditioning. This is done to derive a heuristic for an efficient usage of data reordering techniques and to discuss the achieved results in the context of the impact of rounding error propagation in elementary kernels from Chapter 6 is goal of this chapter.

7.2. Implementation and Platform Issues

Based on the two dimensional block-cyclic data distribution which is explained in Chapter 6.2.2, a parallel CG solver (see Chapter 1.3, Algorithm 9) with Jacobi preconditioning (see Chapter 1.5.1) was implemented.

As software platform, optimized PBLAS routines from the Intel MKL in version 10.1.3.27 have been taken as well as OpenMPI in version 1.4.2, C-XSC version 2.2.3 and the Intel-Compiler in version 11.1.073. In all experimental setups only one MKL thread was used per process in order to avoid additional effects from the fine grained parallelism within the library. All experiments have been performed on the Institutscluster which is located at the Steinbuch Computing Centre (SCC) at the Karlsruhe Institute of Technology (KIT). It consists of 200 computing nodes each equipped with two Intel quadcore EM64T Xeon 5355 processors running at 2,667 GHZ, 16 GB of main memory and an Infiniband 4x DDR interconnect with fat-tree

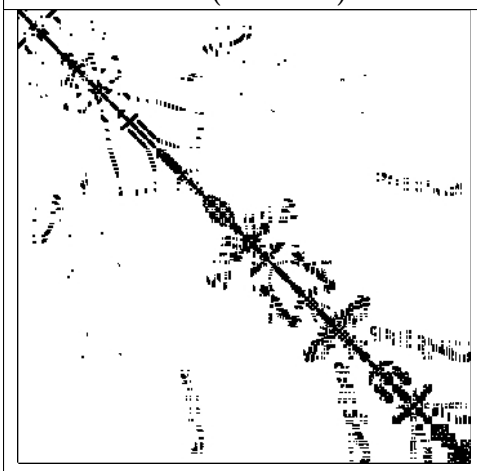
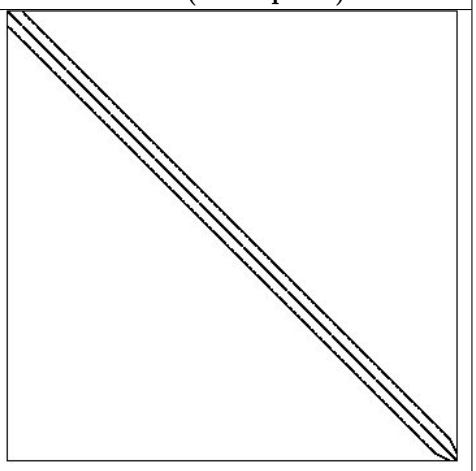
M11 (bcsstk13)	M12 (s1rmq4m1)
	
problem: Fluid flow, FEM problem size: $n = 2003$ sparsity: $nnz = 42943$ cond. number: $4,6 \cdot 10^{10}$ Frobenius norm: $7,5 \cdot 10^{12}$	problem: Cylindrical shells, FEM problem size: $n = 5489$ sparsity: $nnz = 143300$ cond. number: $3,21 \cdot 10^{6}$ Frobenius norm: $1,1 \cdot 10^{5}$

Table 7.1. Sparsity plots and properties of the test-matrices M11 and M12

topology. 17,57 TFlops is the overall peak performance of the whole system and 15,2 TFlops in the Linpack benchmark.

One process was placed on one node and the allocated resources for the experiments were used exclusively to avoid effects in relation to other programs that are executed simultaneously. All results have been computed at least five times to achieve reliable results.

7.3. Numerical Experiments

7.3.1. Implementation Issues and Reference Example. As test-case the matrices bcsstk13 and s1rmq4m1 from matrix market [**Mar**] were chosen. Their characteristics are shown in Table 7.1 and more details can be found on [**Mar**].

For the experiments regarding the accuracy of a matrix-vector multiplication, the vector was filled with pseudo random numbers within the interval $[-0.5, 0.5]$ and the results were verified by using C-XSC, analogue to the previous chapter. As initial guess for the solver a vector with ones in every entry and the zero-vector as right-hand has been chosen.

7.3.2. Numerical Results. Analogue to the previous chapter, the average error (arithmetic mean) to the verified result of an matrix-vector multiplication is shown. Again it can be observed that the more columns the grid of processes has, the better is the accuracy.

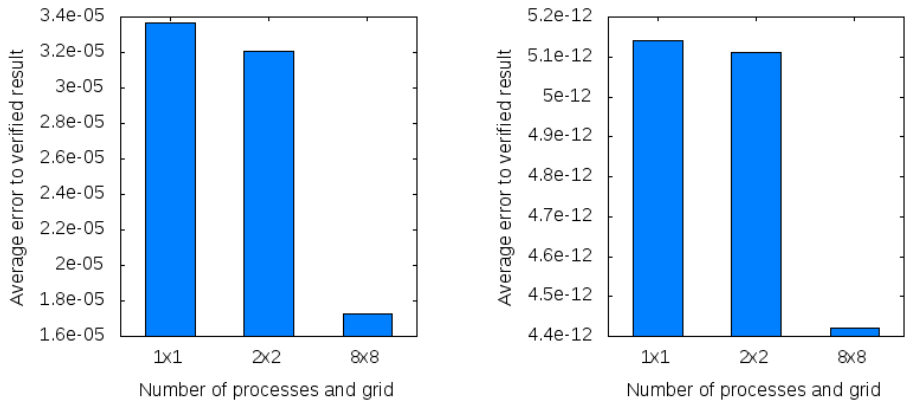


Figure 7.1. Average error to the verified result for three different grids of processes for the test cases M11 (left plot) and M12 (right plot).

In Table 7.2, the best possible accuracy for the error in the A-norm is shown for different grids of processes and the thereby inducted data distribution.

	M11	M12
Grid of processes	Error (A-norm)	
1×1	1.45e-08	2,64e-05
2×2	1.36e-08	2,10e-05
8×8	1.20e-08	8,74e-06

Table 7.2. Shown is the best accuracy that can be reached with a Jacobi-preconditioned CG-solver.

7.4. Result Interpretation

In the previous chapter it is shown from a theoretical and experimental side that the data distribution can have a significant impact on the accuracy of the result for basic linear algebra operations. When such operations are used to build a solver for linear systems, as it is shown in the present chapter, the properties of the linear system that is solved plays, beside the properties of the linear solver and it's implementation, a governing role for the accuracy of the result. For the simple test-cases it is not possible to fall below a certain accuracy when the data distribution is done such that the elementary matrix-vector operations deliver not the most accurate result.

If the linear system is solved with a solving process of e.g. a nonlinear, in-stationary problem, it could be beneficial to dynamically reorder the data depending on the needed accuracy and performance. Empirically, the first steps in a Newton-based solver should be computed highly accurate, while afterwards the accuracy can be reduced. This means that one should start using as many columns in the grid of processes at the beginning and move to a transposed scheme during the solution process. For some cases, this strategy might also be the other way round and the presented results and described approaches shall be understood as indication that even for small systems the effects of rounding errors and communication patterns can be significant.

One important issue should be investigated in future work. While the accuracy of matrix-vector operations is usually better the more columns the grid of processes

has, the accuracy of vector operations (that are based on the same data distribution) can decrease, since the more columns the grid has the fewer rows are involved and vectors are distributed to fewer processes. This leads to an execution scheme for scalar products that is performed in the worst case according to accuracy.

CHAPTER 8

Application Employing New Technologies in Computer Architecture: Simulation of Tropical Cyclones

Results presented in this chapter have been reviewed, presented and published in conjunction with the PARS-workshop 2009, organized by the "Gesellschaft für Informatik e.V.". The related publication is [**HHR09a**].

In this chapter the focus lies on direct numerical simulation (DNS) to solve the governing equations as described in (8.1). 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 (see Chapter 2.1.4 and Figure 2.4) and following generations promise high computational performance, as shown in Chapter 5.2 and 4, at low financial effort and moderate power consumption compared to CPUs from the same generation. In the following it is investigated under which conditions GPUs can be used to accelerate the computation.

The governing equations for the application problems discussed in the following are the time-dependent incompressible Navier-Stokes equations arising from continuum mechanical descriptions of fluid behavior:

$$\rho \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\nabla p + \mu \Delta \mathbf{u} + \mathbf{f}, \quad \nabla \cdot \mathbf{u} = 0, \tag{8.1}$$

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. The meteorological model is based on these equations, completed by initial and boundary conditions.

8.1. The Meteorological Model

An existing meteorological application for simulating tropical cyclones in 2D was considered for acceleration with CUDA (see Chapter 2.2.3). The task is to solve the emerging Navier-Stokes equations (8.1) on a square geometry, as exemplary shown in Figure 8.1. 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 e.g. in [**Bro92, lio03**].

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:

$$\xi = \partial_x v - \partial_y u \quad \text{and} \quad u = -\partial_y \Psi, v = \partial_x \Psi. \tag{8.2}$$

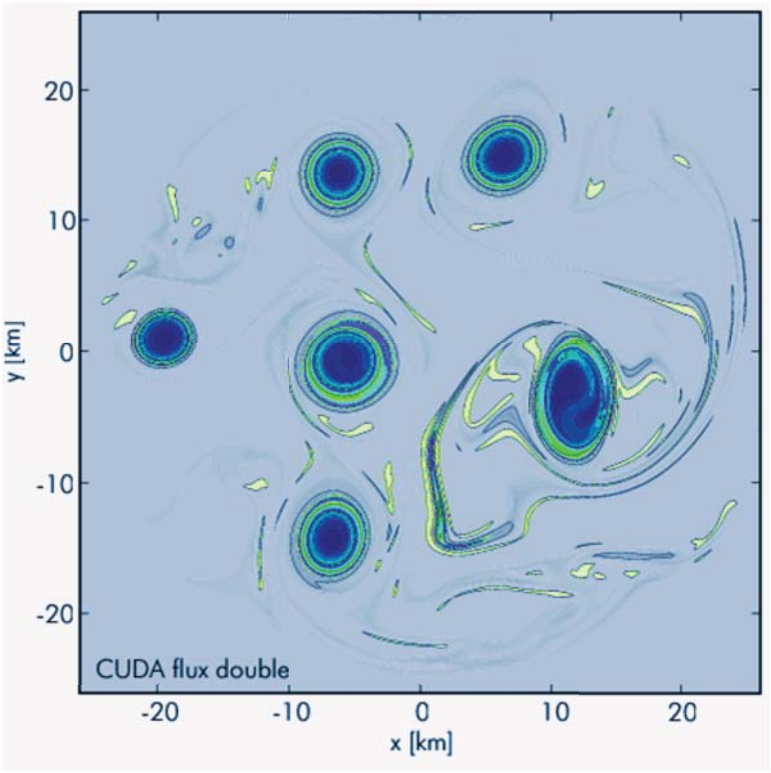


Figure 8.1. Vorticity plot produced by the 2D meteorological application for simulating tropical cyclones. On a square domain the incompressible Navier-Stokes equations are solved in their stream vorticity formulation.

By applying the curl operator on equation (8.1) the pressure term vanishes and with ξ and Ψ as above and $\mathbf{f} = (f_1, f_2)^T$ it remains:

$$-\Delta\Psi = \xi, \tag{8.3}$$

$$\partial_t \xi + u \cdot \partial_x \xi + v \cdot \partial_y \xi - \mu\Delta\xi = \partial_x f_2 - \partial_y f_1. \tag{8.4}$$

The discretization for solving the Poisson equation (8.3) 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 it is possible to compute u and v via (8.2) and update ξ by using equation (8.4). This procedure is then continued iteratively. A 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 [**MB02**].

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 which are very well optimized for the used Intel-based hardware.

8.2. Numerical Experiments

Performance results of the elementary kernels on GPUs are shown in Chapter 5.2.2. In order to reveal the speedup that can be achieved by employing a certain GPU in comparison to a certain CPU, the performance evaluation has been done for

a special hardware constellation. All results in this chapter have been obtained with ASUS ENGTX280 (chip: GT200a) graphics cards on an ASUS P5B deluxe mainboard with Intel P965 north bridge and an Intel Core 2 Duo E6600 CPU. The experiments have been performed by using the libraries MKL 10.0.1.014 and CUBLAS/CUFFT 1.0 using CUDA driver 177.13.

In order to analyze the solver speed for the overall problem the performance of the hardware platform for the elementary kernels *csrmv*, *axpy*, *dot* and *fft* has to be investigated. Since an off-chip (CPU) accelerator is used, bandwidth between host and GPU has also to be examined.

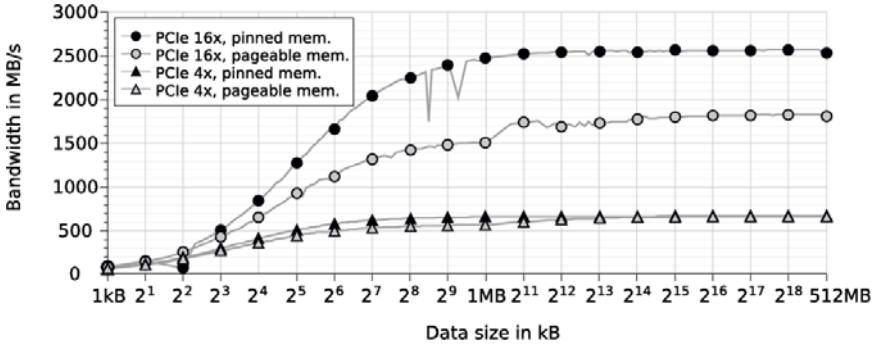


Figure 8.2. Host-to-device memory bandwidth test results.

The north-bridge offers one PCIe slot connected with 16 lanes and another one with 4 lanes only. As plotted in Figure 8.2 one can achieves approximately 2.5 GB/s on the faster slot and 0.65 GB/s on the other. 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.

For the vector operation *saxpy* (single precision) and *daxpy* (double precision) the GPU can clearly outperform the CPU for large vector dimension (Figure 8.3). Due to a lack of data re-use (low computational intensity) 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* so close, that another plot would have been superfluous. Working exclusively on the GPU with large vectors is the key to achieving high speed-ups over MKL on a conventional CPU.

For the *csrmv* 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 8.4).

The measurements of the 2D FFT are done by forward and backward real transforms. Although the plots show only speed-ups of 1.2 and 1.3, the complete meteorologic problem solver demonstrates that in combination with other kernels, speed-ups increase significantly.

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 8.5, just minor improvements could be achieved.

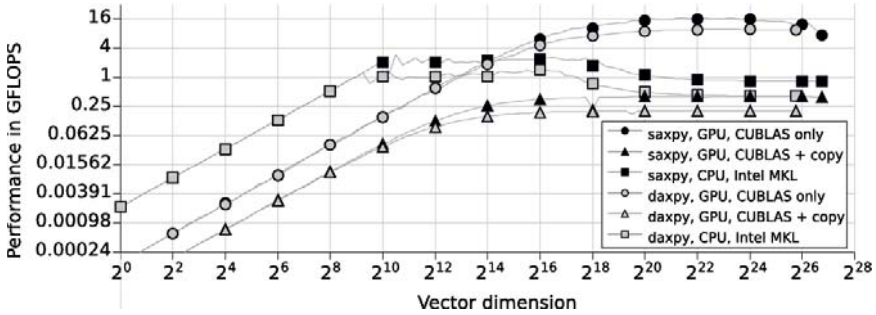


Figure 8.3. Performance test results of the *axpy* kernel in single and double precision.

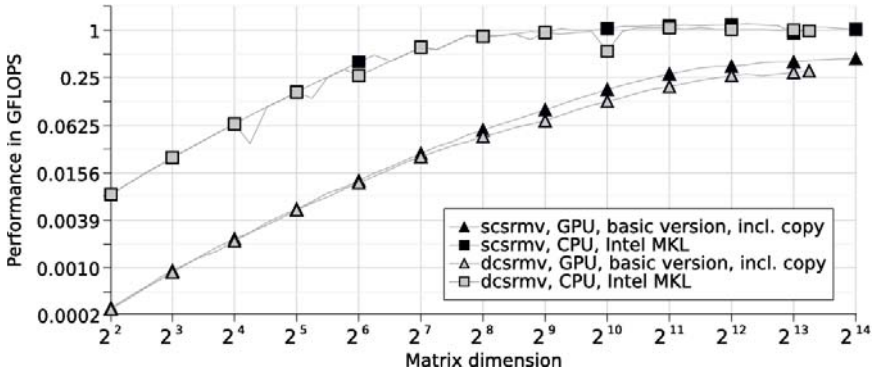


Figure 8.4. Performance test results of the *csrmv* kernel in single and double precision.

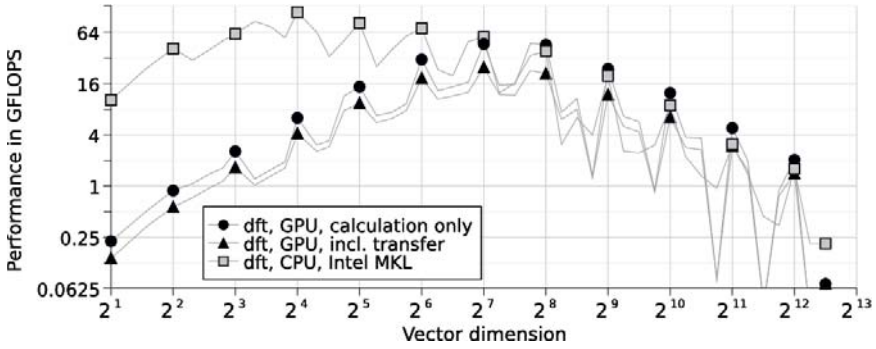


Figure 8.5. Performance test results of the 2D real-to-complex *fft* in single precision.

It is to be underlined, that all tests could solely be done with CUFFT 1.0. Newer 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 2.4). 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 8.6.

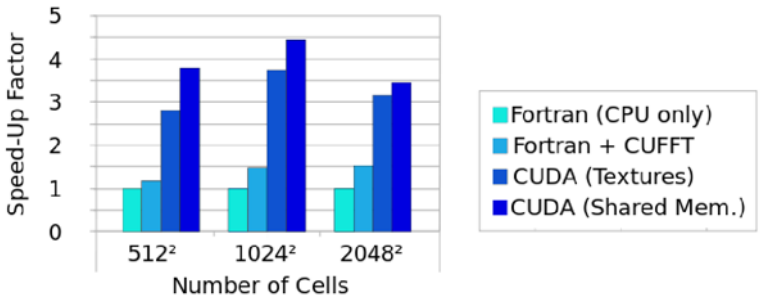


Figure 8.6. Performance improvements using GPU acceleration. Speed-ups of the 2D meteorological application based on the number of discretization points in any direction.

8.3. Conclusion

The current as as previous GPU generations offer enormous potential that can be utilized not only in constructed examples but also CFD applications with academic and non-academic background. A basic condition for this is that the underlying mathematical model combined with the numerical schemes for solving it offers enough parallelism to create enough threads for the GPU to cover waiting time for memory calls of one GPU-thread by executing another thread. Exchanging threads is for free compared to the time a memory operation takes.

The programmability of NVIDIA-GPUs was heavily simplified by the introduction of CUDA compared to the most common former approaches. Due to that, 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 the partially ported meteorological implementation shows. To port a code completely to the GPU then offers the full performance of the architecture. In the future, OpenCL might macerate the tradeoff that achieving high performance often means to loose portability (as it is the case for CUDA implementations where no previous CPU-implementation is available).

All this is available for a relatively low prize not only for consumer graphics cards but also as professional solutions for high performance computing. By looking to the TOP500 list [**top10**] the number of clusters with accelerators, often NVIDIA GPUs, increases more and more and gives a strong impulse to make use of this technique. Additionally, the performance per Watt ratio is very good for such machines, as Chapter 5.2 and the Green500 list [**gre10**] indicate.

CHAPTER 9

Summary and Outlook

Within this last chapter, the content of the thesis, main results and open questions are addressed in an aggregated form. Additionally, possible variants of further research and developments are depicted.

Goal of the thesis was to analyze how modern hardware technologies can be used to accelerate numerical simulations. Special focus was set on accuracy and performance for solvers for linear systems of equations in the context of finite element discretizations of partial differential equations. As résumé it can be stated that within this thesis it is shown that there are numerical methods that can take great advantage of the properties modern hardware offer in both, accuracy as well as performance.

A short survey of the developments in modern hardware architectures opens the thesis. Explained is the growth of performance within the last decade and todays situation, where there are machines that can execute more than 10^{15} floating point operations per second (FLOPS). Parallelism is identified as the strategy to build such computers and the enormous complexity that comes along with hundred thousands of cores and arising heterogeneous hardware platforms (employment of GPUs and other coprocessor technologies) constitutes a big challenge for efficient numerical simulations. Applications requesting extensive computational resources can be found in many areas. Especially solvers for sparse linear systems arising in the area of computational fluid dynamics frequently show a poor scalability since the problems are often fully coupled and have a large number of unknowns. The need for efficient linear solvers can also be derived from the large number of projects aiming at this topic or the numerousness of software libraries that are developed in research institutions all over the world (e.g. MUMPS, Pardiso, PETSc, ScaLAPACK, Trilinos and many others).

The Conjugate Gradient method (CG), as well as the Generalized Minimum Residual method (GMRES), are two common solvers for linear systems of equations. Both are explained in **Chapter 1.1** as well as preconditioning techniques in form of (Block-) Jacobi and incomplete LU-factorization. Because of their strong dependence on the application, only basic strategies for preconditioning are shown. Further developments aiming particularly at preconditioning should take into account the usage of fix-point and staggered arithmetic since upcoming technologies in reconfigurable computing (FPGAs) seem very suitable for such techniques. When nonlinear problems are solved, it seems to be beneficial to discuss the strategy of choosing the stopping criteria of the linear solver dynamically (by assuming a linearization step within the nonlinear solver like it is the case within Newtons method).

Chapter 2 gives an overview of modern (2010) technologies in hardware architecture. First, basic information about parallel micro- and multiprocessor systems is given, followed by a closer look onto graphic processing units (GPUs). In the area of computational fluid dynamics, field programmable gate arrays (FPGAs) are an emerging architecture which is also described with special focus on a dedicated

hybrid CPU-FPGA machine. To use parallel architectures, special programming paradigms (respectively APIs) have to be employed since automatic parallelization of pure sequential code through the compiler is usually not efficient. OpenMP, as a representative of paradigms for a global address space (also called shared memory), MPI, which assumes a distributed address space (or distributed memory), as well as CUDA for certain GPUs, OpenCL and PGAS languages are explained. OpenMP, MPI and CUDA are heavily used in the numerical experiments of the following chapters. OpenCL and PGAS languages are not evaluated, but should be taken into account in future studies. For the future is can be stated that paradigms should consider data locality as main issue (as it is done in PGAS languages).

A promising approach to exploit the hardware capabilities of present and in the medium term arising heterogeneous high performance computers is the iterative refinement method in combination with mixed precision techniques as explained in **Chapter 3.1**. This is true at least for the reference examples examined in Chapter 5.4 from the area of computational fluid dynamics as well as for most of the artificially created test-matrices. One future step could be to analyze how the perturbation of the matrix, that occurs when it is transformed from one floating point format to another, effects the condition number and if it could be beneficial to assemble the matrices in the needed floating point format instead of converting it as done in this work. The numerical analysis regarding mixed precision techniques applied to elliptic operators is, as far as the author knows, never published before. Analytical boundaries are presented to guarantee that the method error does not exceed the error that comes along within the discretization step. Based on the presented results, multiple promising opportunities are possible. A posteriori error estimators can be employed to approximate the necessary constants. Hereby it is very interesting that, depending on the error estimator, local information about the error is available which could be used to choose the necessary accuracy within the solution process locally. Deducted algorithms can be assumed to be very suitable to be mapped on heterogeneous hardware, offering properties that correspond to the needs of the part of the algorithm. Also imaginable are implementations varying the floating point formats based on iteratively computed bounds for the perturbations during runtime. Suitable libraries (such as the GNU Multi Precision library GMP) or hardware that offer a free choice of working on arbitrary floating point formats (like FPGAs) are already existing and the challenge is to develop suitable numerical methods.

Hardware-aware numerical mathematics improves performance in many cases, as shown in the evaluation of different implementations of the CG and GMRES solvers based on dedicated hardware from **Chapter 4**. It is demonstrated how both above mentioned solvers can be implemented in parallel using the paradigms OpenMP and MPI on CPU-based architectures and CUDA for GPUs. It is shown that by using special communication patterns the structure of the underlying linear system can be exploited leading to significantly gains in scalability. Especially the proposed neighbor communication was seen to be beneficial in some cases. Additionally, the employment of software libraries offering highly optimized BLAS implementations, as well as complete linear solvers, is addressed to reveal when it is beneficial to deploy them. Based on multiple sparse linear systems arising from different applications it is shown that parallelization significantly reduces the computation time. The results also indicate constrains for the scalability and the need to develop more advanced linear solvers for parallel and hybrid architectures. In future research, the flexible GMRES should be taken into account in combination with preconditioning techniques that automatically adapt to the properties of the underlying problem while simultaneously taking into account the characteristics of the underlying hardware.

Every hardware, and every future hardware generation, has different and sometimes new kinds of characteristics. In order to develop hardware-aware solvers, detailed knowledge about the properties of the hardware and behavior of relevant solvers is indispensable. Prototypical evaluations of CPU-based clusters, GPUs as well as FPGAs are presented in **Chapter 5** and can be seen as a reference scheme for other researchers. All "classes" of architectures are considered in particular, followed by benchmarks that compare the different architectures with each other in terms of performance as well as in terms of energy efficiency. As a result it can be stated that GPUs are capable to solve sparse linear systems significantly faster then single CPUs. But with respect to energy efficiency, respectively costs per solution, the user has to define how expensive the results can be. It becomes obvious that in the (near) future, adequate metrics have to be defined to measure the costs of a solution not only with respect to performance but also with respect to other criterias like energy efficiency, accuracy, reliability and so on. FPGAs and the software to support the usage of high-level programming languages on such architectures has to be strongly improved as the presented results indicate. This is also true for the numerical methods that can be executed on such machines. Topics, which should be covered in future research, are surely the usage of interval arithmetics, staggered arithmetics and other methods that can be implemented in hardware instead of running on a higher abstraction level. Preconditioners that work on integers could be a first step into this direction and could extend the analytical work from Chapter 3.1.

Empirical observations document that different data distributions can lead to different results in numerical simulations. This problem becomes worse on parallel and heterogeneous hardware and is very important to be discussed. Beside the theory of rounding error propagation for elementary operations and multiple data distributions, experiments validating the theoretical findings are shown in **Chapter 6**. Based on the two-dimensional block-cyclic data distribution, routines from the PBLAS/ScaLAPACK library have been evaluated with focus on accuracy and performance for different test-cases. As reference for the quality of the computed results, verified computing was utilized in the form of the C-XSC library. Important in this set-up is that the choice of the linear system and right-hand side is no longer limited to examples where the analytical result is known (to validate the results). Arbitrary linear systems and right hand sides can be used and all results can be verified. This has never been published before in such a constellation, as far as the author knows. The theoretical results can be applied to many sparse data formats and demonstrate that the more columns the grid of processes has the better is the upper bound for the rounding error propagation. In the presented numerical experiments, the accuracy for almost all test-cases show more accurate results for increasing number of columns. One order of magnitude and more can be gained. Depending on the topology of the communication layer (usually determined by the hardware structure of the interconnect of the cluster), the performance behaves contrarily due to imbalances in the communication scheme. Based on the shown results it is possible to implement a linear algebra, respectively solvers, that can reorder the data during runtime to optimize for speed and/or accuracy.

As canonical continuation, the impact of the data distribution on the convergence of a parallel, preconditioned CG solver was evaluated in **Chapter 7**. It is shown for two test-cases that a higher accuracy within parts of the underlying linear algebra lead to more accurate results for the solution of the linear system. Here, detailed numerical analysis of the solver with respect to rounding errors could help figuring out the impact on the convergence rate of every single operation within the algorithm. In addition to that, further numerical experiments should be performed and more preconditioners should be taken into account.

Chapter 8 presents a model for simulations of tropical cyclones, which is in everyday use by scientists from meteorology. The application was ported from a pure CPU-based implementation to a CUDA-based simulation which can be executed on GPUs. Beside three implemented versions running exclusively on one or the other architecture, a hybrid implementation employing both, CPU and GPU, was created since parts of the simulation could easily be shifted to a GPU. A speedup of close to five is the result for the overall simulation on relevant grid sizes when the optimized GPU-implementation (employing texture-caches) is used. Users of the simulation are now able to create almost five times more results in the same time they created only one result before, or in other words, they need only about 20 percent of the time to perform a simulation.

In the midterm, lots of heterogeneous hardware architectures will appear. They will be composed of conventional CPUs in combination with GPUs (on- and off-die), FPGAs (on- and off-die) as well as other kinds of accelerator technologies. Programming those machines based on a single, wide spread programming paradigm (or API) seems only possible by using OpenCL. It shall be annotated that by using this programing paradigm, implementations are more portable then with most other paradigms but in order to exploit the available performance on a dedicated hardware, a special tailoring of the code to the machine has to be performed leading potentially to a decrease in performance (or even faults) on other systems. Data locality might play the most important role in the future, since even today the energy spend within the computation is mainly spend for the transportation of the data, not for performing arithmetic operations. The efforts to integrate additional functionalities to the MPI standard may also be observed carefully in the future.

Reproducibility of numerical results is a basic request from scientific researchers. Since there exists hardware (e.g. FPGAs) offering the possibility to execute the needed operations directly in hardware, verified computing may play an important role in the future. This is especially the case in the area of dependable computing, where at least some parts (or results) of an algorithm have to be computed with a guaranteed accuracy. However, all future developments will be analyzed carefully in terms of energy efficiency since more and more energy is spend for computing power and hardware vendors will examine strictly the utilization of transistors.

All results presented in this work express the inspiration to give a contribution to the scientific community in order to solve problems arising in the area of numerical simulation in a more efficient way. Last, but not least, it is stated that the holistic view of this work is motivated by the idea that a problem should never be solved without spending a blink of an eye on the whole.

Bibliography

[AAB+10] H. Anzt, W. Augustin, M. Baumann, H. Bockelmann, T. Gengenbach, T. Hahn, V. Heuveline, E. Ketelaer, D. Lukarski, A. Otzen, S. Ritterbusch, B. Rocker, S. Ronnas, M. Schick, C. Subramanian, J.-P. Weiss, and F. Wilhelm. Hiflow3 - a flexible and hardware-aware parallel finite element package. EMCL Preprint Series, 2010.

[ABV10] H. Anzt, Rocker B., and Heuveline V. An error correction solver for linear systems: Evaluation of mixed precision implementations. In *EMCL Preprint Series*, 2010.

[ACF+11] H. Anzt, M. Castillo, J. C. Fernández, V. Heuveline, R. Mayo, E. S. Quintana-Ortí, and B. Rocker. Power Consumption of Mixed Precision in the Iterative Solution of Sparse Linear Systems. EMCL Preprint Series, 2011.

[ADR92] M. Arioli, I. Duff, and D. Ruiz. Stopping criteria for iterative solvers. *SIAM J. MATRIX ANAL. APPL.*, 13(1):138–144, January 1992.

[AH74] G. Alefeld and J. Herzberger. *Einführung in die Intervallrechnung*. Springer-Verlag, Heidelberg, 1974.

[AH83] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983.

[AHHRed] H. Anzt, T. Hahn, V. Heuveline, and B. Rocker. GPU Accelerated Scientific Computing: Evaluation of the NVIDIA Fermi Architecture; Elementary Kernels and Linear Solvers. In *Proceedings of HipHaC 2011*, accepted.

[AHR] H. Anzt, V. Heuveline, and B. Rocker. Mixed precision error correction methods for linear systems - convergence analysis based on krylov subspace methods.

[AHR11] H. Anzt, V. Heuveline, and B. Rocker. An error correction solver for linear systems: Evaluation of mixed precision implementations. In J. Palma, M. Dayd, O. Marques, and J. Lopes, editors, *High Performance Computing for Computational Science - VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 58–70. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19328-68.

[AHW10a] W. Augustin, V. Heuveline, and J.-P. Weiss. Convey HC-1 – The Potential of FPGAs in Numerical Simulation, 2010.

[AHW10b] W. Augustin, V. Heuveline, and J.-P. Weiss. Convey HC-1 – The Potential of FPGAs in Numerical Simulation, 2010.

[ALO02] G. Alefeld, I. Lenhardt, and H. Obermaier. *Parallele numerische Verfahren*. Springer, Berlin, Heidelberg, New York, 2002.

[ARH10] H. Anzt, B. Rocker, and V. Heuveline. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms - an evaluation of different solver and hardware configurations. *Computer Science - R&D*, 25(3-4):141–148, 2010.

[BBC+94] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 2 edition, 1994.

[BBC+08] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, W. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor and study lead, 2008.

[BBD+08] M. Baboulin, A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszcek, J. Kurzak, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. 2008.

[BBE+08] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[BBG+09] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.

[BCC+96] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed memory computers Design Issues And Performance. In *SUPERCOMPUTING '96*. IEEE Computer Society, 1996.

[BDD+02] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[BDK+08] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34:17:1–17:22, July 2008.

[BDL+07] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszcek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. 2007.

[BES98] A. Björck, T. Elfving, and Z. Strakos. Stability of conjugate gradient and lanczos methods for linear least squares problems. *SIAM J. Matrix Anal. Appl.*, 19:720–736, July 1998.

[BFGS03] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22:917–924, July 2003.

[BGMS97] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[Bra07] D. Braess. *Finite Elemente - Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*, volume 4.,überarbeitete und erweiterte Auflage. Springer, Berlin Heidelberg New York, 2007.

[Bre10] T.M. Brewer. Instruction set innovations for the convey hc-1 computer. *Micro, IEEE*, 30(2):70 –79, mar. 2010.

[Bro92] R. A. Brown. *Fluid Mechanics of the Atmosphere*. Academic Press, London, GB, 1992.

[CJvdP08] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: portable shared memory parallel programming*. Scientific and Engineering Computation. MIT Press, Cambridge, Mass., 2008.

[Con05] UPC Consortium. Upc language specications v1.2. Technical report, Lawrence Berkeley National Lab, 2005.

[CP09] X. Chen and KK. Phoon. Some numerical experiences on convergence criteria for iterative finite element solvers. *Computers and Geotechnics*, 2009. doi:10.1016/j.compgeo.2009.05.012.

[Dav] T.A. Davis. The university of florida sparse matrix collection. Technical report, University of Florida. http://www.cise.ufl.edu/research/sparse/matrices.

[DD05] Jim Demmel and Jack Dongarra. LAPACK 2005 prospectus: Reliable and scalable software for linear algebra computations on high end computers. LAPACK Working Note 164, feb 2005. UT-CS-05-546, February 2005.

[DDH07] J. Demmel, I. Dumitriu, and O. Holtz. Fast linear algebra is stable. *Numer. Math.*, 108(1):59–91, 2007.

[DHK+06] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Softw.*, 32(2):325–351, 2006.

[DL05] J. Dongarra and P Luszczek. Introduction to the HPC Challenge benchmark suite. Technical Report UT-CS-05-544, University of Tennessee, 2005.

[DLP03] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[DRDSA+95] J. Drkosova, M. Rozloznk, J. Drko Sov A, M. Rozlo Zn Ik, Z. Strakos, and A. Greenbaum. Numerical stability of gmres, 1995.

[DW93] J. Dongarra and D. Walker. Lapack working note 58: The design of linear algebra libraries for high performance computers. Technical Report UT-CS-93-188, Knoxville, TN, USA, 1993.

[EGPG96] Carter Edwards, Po Geng, Abani Patra, and Robert Van De Geijn. Parallel matrix distributions: Have we been doing it all wrong?, 1996.

[Fac00] A. Facius. *Iterative solution of linear systems with improved arithmetic and result verification*. PhD thesis, University of Karlsruhe (TH) - Institut für Angewandte Mathematik, 2000.

[fC] Steinbuch Center for Computing. `http://www.scc.kit.edu`.

[Fly72] M.J. Flynn. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[GalXX] G. P. Galdi. An introduction to the mathematical theory of the navier-stokes equations, 19XX.

[Gal10] B. Galler. Parallel programming paradigms in the context of fluid dynamics, 2010. Diploma-thesis, adviced by Heuveline, V. and Rocker, B.

[GG05] Maya B. Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, Dordrecht, Netherlands, 2005.

[GHW06] L. Giraud, A. Haidar, and L. T. Watson. Mixed-precision preconditioners in parallel domain decomposition solvers. Technical Report TR/PA/06/84, CERFACS, Toulouse, France, 2006. Also appeared as IRIT Technical report ENSEEIHT-IRIT RT/APO/06/08.

[GK95] A. Gupta and V. Kumar. Parallel algorithms for forward elimination and backward substitution in direct solution of sparse linear systems. In *Direct Solution of Sparse Linear Systems, Proceedings of the Supercomputing Conference 1995*, 1995.

[GLLS03] N. Goodnight, G. Lewint, D. Luebke, and K. Skadron. A multigrid solver for boundary-value problems using programmable graphics hardware. In *In Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 102–111, 2003.

[gre10] Website of the green500 project., December 2010.

[Gro07] H.-G. Gromann, C. ; Roos. *Numerical treatment of partial differential equations*. Universitext. Springer, Berlin, 2007.

[GS05] D. Groth and T. Skandier. *Network+ Study Guide*. Sybex Inc., 2005.

[GS08] D. Göddeke and R. Strzodka. Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations (part 2: Double precision gpus). Technical report, Fakultät für Mathematik, TU Dortmund, 2008.

[GST07] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations. 2007.

[Hac10] W. Hackbusch. Elliptic differential equations : Theory and numerical treatment, 2010.

[Hah09] T. Hahn. Numerical simulation of 3d particulate flows based on gpu technology. Master's thesis, 2009. Diploma-thesis, adviced by Heuveline, V. and Rocker, B.

[HHR09a] T. Hahn, V. Heuveline, and B. Rocker. GPU accelerated scientific computing: Fluid and particulate flows with CUDA. In GI/ITG-Fachgruppe PARS, editor, *Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen*, volume 26, pages 135–144, 2009.

[HHR09b] T. Hahn, V. Heuveline, and B. Rocker. Gpu-based simulation of particulate flows with cuda. In *Proceedings of the PARS Workshop 2009*. German Informatics Society, 2009.

[Hig96] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics - SIAM, Philadelphia, PA, USA, 1996.

[HKL09] V. Heuveline, M.J. Krause, and J. Latt. Towards a hybrid parallelization of lattice boltzmann methods. *Computers and Mathematics with Applications*, 58:1071–1080, 2009.

[HL08] S. Hoffmann and R. Lienhart. *OpenMP: Eine Einführung in die parallele Programmierung mit C/C++*. Springer, Berlin, Heidelberg, 2008.

[HP07] J.L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, Amsterdam, 4th edition, 2007.

[HP09] J.L. Hennessy and D. Patterson. *Computer Organization and Design*. Elsevier, Amsterdam, 4th edition, 2009.

[HPC] HPCC. HPCC Results Database. `http://icl.cs.utk.edu/hpcc/hpcc_results.cgi`.

[HRKH97] R. Hammer, D. Ratz, U. Kulisch, and M. Hocks. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems*. Springer-Verlag New York, Secaucus, NJ, USA, 1997.

[HRR09] V. Heuveline, B. Rocker, and S. Ronnas. Numerical Simulation on the SiCortex Supercomputer Platform: a Preliminary Evaluation. EMCL Preprint Series, 2009.

[ilu] *ILU++*. http://iamlasun8.mathematik.uni-karlsruhe.de/∼ae04/iluplusplus.html.

[KK99] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[KKL+93] R. Klatte, U. Kulisch, C. Lawo, R. Rauch, and A. Wiethoff. *C-XSC- A C++ Class Library for Extended Scientific Computing*. Springer-Verlag Berlin, 1993.

[KM81] U. Kulisch and W. L. Miranker. *Computer Arithmetic in Theory and Practice*. Academic Press, New York, 1981.

[KW03] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22:908–916, July 2003.

[LAP] LAPACK. Linear Algebra Package. http://www.cs.colorado.edu/ jessup/lapack/. visited in 09th February 2009.

[LBM] LBMethod.org. `http://www.lbmethod.org`.

[Lib] The GNU Multiple Precision Arithmetic Library. `http://gmplib.org`.

[Lin70] P. Linz. Accurate floating-point summation. *Commun. ACM*, 13(6):361–362, 1970.

[lio03] Handbook of numerical analysis, 2003.

[LLL+06] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 113, New York, NY, USA, 2006. ACM.

[MAK03] R. C. Mittal and A. H. Al-Kurdi. An efficient method for constructing an ilu preconditioner for solving large sparse nonsymmetric linear systems by the gmres method. *Computers and Mathematics with Applications*, (45):1757–1772, 2003.

[Mar] Matrix Market. `http://math.nist.gov/MatrixMarket`.

[May] J. Mayer. Symmetric permutations for i-matrices to delay and avoid small pivots during factorization. To appear in SIAM J. Sci. Comput.

[May07] J. Mayer. A multilevel crout ilu preconditioner with pivoting and row permutation. *Numerical Linear Algebra with Applications*, (14):771–789, 2007.

[MB02] A. J. Majda and A. L. Bertozzi. Vorticity and incompressible flow. *ambridge Texts in Applied Mathematics*, page chapter 2.2, 2002.

[Mes09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*, September 2009. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.

[MLID09] Brian J. Martin, Andrew J. Leiker, James H. Laros III, and Doug W. Doerfler. Performance analysis of the sicortex sc072. The 10th LCI International Conference on High-Performance Clustered Computing, 2009.

[MLT98] P. J. Mucci, K. London, and J. Thurman. The CacheBench Report. Technical report, Innovative Computing Laboratory, University of Tennessee, 1998.

[MLT99] P. J. Mucci, K. London, and J. Thurman. The BLASBench Report. Technical report, Innovative Computing Laboratory, University of Tennessee, 1999.

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.

[NVI08] NVIDIA Corp. *NVIDIA GeForce® GTX 200 GPU Architectural Overview*, May 2008. `http://www.nvidia.com/object/io\_1213615494642.html`.

[NVI09] NVIDIA Corp. *NVIDIA CUDA™ Programming Guide*, August 2009. `http://developer.download.nvidia.com`.

[Ope] OpenLB. `http://www.openlb.org`.

[Ope08] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[oT] Karlsruhe Institute of Technology. `http://www.kit.edu`.

[RKH11] B. Rocker, M. Kolberg, and V. Heuveline. The impact of data distribution in accuracy and performance of parallel linear algebra subroutines. In J. Palma, M. Dayd, O. Marques, and J. Lopes, editors, *High Performance Computing for Computational Science - VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 394–407. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19328-636.

[RR07] T. Rauber and G. Rünger. *Parallele Programmierung*. Springer, Beelin, Heidelberg, 2nd edition, 2007.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[SB05] J. Stoer and R. Bulirsch. *Numerische Mathematik 2*. Springer, Berlin Heidelberg New York, 2005.

[Sch] Joerg Schulenburg. Compute-Server Kautz. `http://www-e.uni-magdeburg.de/urzs/kautz`.

[Sch10] M. Schmidtobreick. Numerical methods on reconfigurable hardware using high level programming paragigms. Diploma thesis adviced by v. heuveline, KIT, 2010.

[SG06] R. Strzodka and D. Göddeke. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In *IEEE Proceedings on Field–Programmable Custom Computing Machines (FCCM 2006)*. IEEE Computer Society Press, 2006.

[SH96] M. Sidani and B. Harrod. Parallel matrix distributions: Have we been doing it all right? Technical report, Knoxville, TN, USA, 1996.

[She94] J.R. Shewchuk. An introduction to the conjugate gradient method. `http://www.cs.cmu.edu/\~quake-papers/painless-conjugate-gradient.pdf`, 1994.

[SiC] SiCortex. High Capability System : SC5832. `http://sicortex.com/products/high_capability_system_sc5832`.

[SK06] H.R. Schwarz and N. Köckler. *Numerische Mathematik*, volume 6. Teubner, Wiesbaden, 2006.

[SS86] Y. Saad and M. H. Schultz. Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986.

[top10] Website of the top500 project, December 2010.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

Index