Asynchronous and Multiprecision Linear Solvers

Scalable and Fault-Tolerant Numerics for Energy Efficient

High Performance Computing

Zur Erlangung des akademischen Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

von der Fakultät für Mathematik des
Karlsruher Instituts für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-Math. techn. Hartwig Anzt

aus
Karlsruhe

Tag der mündlichen Prüfung: 28. November 2012

Referent:	Prof. Dr. Vincent Heuveline
Korreferent:	Prof. Dr. Jack Dongarra
Korreferent:	Prof. Dr. Rudolf Lohner

Abstract

The development of modern technology is characterized by simulations, that are no longer performed in physical experiments, but in terms of mathematical modeling by partial differential equations. Numerical algorithms have often replaced the necessity of laboratories, but typically demand for immense computing power. While the hardware manufacturers try to keep up with the demand for Petaflops, the suitability of the numerical methods employed in the simulation algorithms decreases constantly. This often stems from a gap between the design and parallelism of the numerical algorithms forming the simulation code and the parallelism and complexity provided by today's and future hardware platforms, impacting performance, dependability and resource efficiency.

In a nutshell, three main challenges can be identified when aiming for exascale simulation algorithms: scalability, reliability and energy efficiency. As future hardware architectures are expected to consist of several millions up to billions of processing units located in different devices connected via different communication technologies, the algorithms running on these machines efficiently are required to scale for this immense processor number, which implies the reduction of the communication to a minimum. Furthermore, as the high number of processors also implies a significant failure rate, a high tolerance to hardware error is essential to ensure the completion of the simulations. While checkpointing strategies are widespread used in today's implementations, algorithms will no longer be able to rely on this technique as soon as the hardware complexity induces a mean time of failure of the full system smaller than the time for checkpointing and restarting. Finally the power demand of computing facilities handling the simulation experiments can be identified as a major hurdle. Already today, the energy costs often exceed the acquisition costs after few years posing an economical challenge, and moreover question the power demand and the ecological footprint the resource efficiency of computer simulations. While future hardware is expected to reduce the power demand by featuring efficient accelerator technology and energy saving mechanisms, conventional software usually ignores this issue by allowing only very limited usage of these techniques.

In many simulation applications, generating solution approximations of discretized partial differential equations is the computationally most expensive part in the algorithm - particularly since the traditional numerical methods require both communication and synchronization that limit the efficient hardware usage. In this thesis we target the inevitable question of how numerical solvers can be adapted to future computing facilities by proposing unconventional methods, suitable for the highly parallel and hybrid hardware platforms that are expected for the near future. Especially, we address the topic of hardware-adapted methods by aiming for synchronization-free linear solvers that minimize idle times by removing synchronization barriers, and therefore allow the efficient usage of computer systems consisting of components with different hardware characteristics. The implied high tolerance with respect to communication latencies improves the fault tolerance of the simulation method. As asynchronous methods also enable the usage of the power and energy saving mechanisms provided by the hardware, they address all challenges we identified for numerical methods in the exascale era and combine the most important characteristics required for hardware-efficient simulation algorithms. From the theoretical point we investigate the derived methods with respect to their convergence properties and analyze the potential of adapting them to a specific problem by accounting for the discretization method or the matrix characteristics. Also, we provide a comprehensive study revealing excellent performance, scalability and fault-tolerance properties as well as remarkable energy-efficiency of block-asynchronous iteration on different hardware architectures.

Acknowledgements

I am especially grateful to Prof. Dr. Vincent Heuveline who guided me already through my graduate studies, my diploma thesis and now this work. Particularly, I want to thank him for his positive encouragement, his support, the helpful discussions throughout the last years, but also the academic freedom he offered me while working at the "Institute for Numerical Simulation, Optimization and High Performance Computing". Within this group, the continuous exchange with my colleagues has been beneficial contribution at all times for which I am very thankful.

I am also very grateful to my second advisor Prof. Dr. Jack Dongarra who not only offered me a research visit at the "Innovative and Computing Lab" at the University of Tennessee, but also provided me with useful ideas and hints. In this context, I also would like to thank the Karlsruhe House of Young Scientists (KHYS) for their financial and organizational support enabling this scientific exchange.

Special thanks go to Prof. Dr. Rudolf Lohner for his self-sacrificing support and his diligent corrections in this work.

Furthermore, I want to thank all partners I worked together with on the different research topics. Particularly, I want to mention Prof. Enrique S. Quintana-Ortí from the University of Jaume I in Spain and Prof. Stanimire Tomov from the University of Tennessee, for many fruitful discussions throughout the past years.

Last but not least, I would like to thank all my family and friends for their less scientific, but nevertheless equally important support and encouragement during my venture in the research world.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

1.1. Motivation

In many scientific fields, research and development is characterized by experiments, that are no longer performed in laboratories, but by means of mathematical modeling and numerical simulations conducted on high-performance clusters. In many engineering areas, the complexity of the experiments exceeds the laboratories' capacity, but they may be evaluated in terms of scientific simulations that provide a virtual reality to investigate the processes. Nevertheless, the elation about the efficient usage of computer simulations is confined by the increase in the gap between the numerical methods typically used in the simulation and the target hardware architectures, as past methods hardly accommodate the hardware developments towards highly parallel and heterogeneous systems. The challenge of optimizing the execution of scientific applications on the most recent hardware systems is still open, and this work aims at providing significant advance in addressing this topic.

1.1.1. Hardware Evolution on the Road to Exascale

Within the last decades, the hardware architectures for supercomputing have undergone a process of continuous change asking for the redesign of the numerical algorithms with each new hardware generation[1] [All01, Ber05, MVM09, Rya11].

For almost three decades between 1975 and 2005, supercomputing was mostly dominated by uni-core processors (comprising from vector processors as those used in the CRAY architecture [Rus78] to commodity superscalar CPUs). While the chips contained one single core and the parallelism was realized by the gathering a few, the systems grew in performance mostly by increasing the complexity of the core architecture, such that they were able to execute a single task with increasing speed. When targeting one single processor no changes to the sequential codes were necessary, and the speedup was gained by faster single-core execution. On the other hand, when the target architecture contained multiple uni-core processors, the applications were parallelized using machine-dependent annotations and a parallelizing compiler. Moore's law [Sch97] was suitable to describe the performance improvements during this uni-core period.

Around 2005, hardware developers hit the barriers of this trend: The shrinking of the transistor size, the gathering of transistors in one core, and the growth in clock rate, which

[1] See `http://herbsutter.com/` for a vivid description of the hardware evolution process.

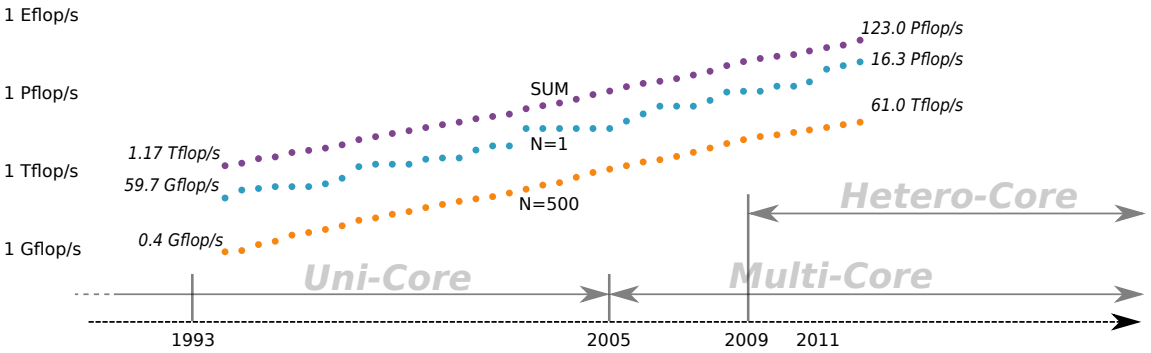


Figure 1.1.: Overview of the trends in hardware design and the performance development. N=1 and N=500 denote the systems ranked No.1, and No.500, respectively, in the TOP500 list [top]. SUM aggregates the computing power of all listed systems.

had so far been the basic principles of accelerating the processors, were no longer beneficial due to effects like thermal restrictions, leakage currents and energy considerations [KBD10]. Particularly the limitation in clock rate triggered a trend towards more parallelism, and instead of designing faster and more complex single-core chips, the hardware developers started to aggregate multiple cores into one chip - the beginning of the *Multicore-Era*, see Figure 1.1. The performance gains of this trend still fit Moore's law, but exploiting the computational power became much more difficult. The software developers had to redesign the code, often even to reformulate their algorithms and applications into parallel programs that could make efficient use of the parallel hardware. For applications that already have lots of inherent parallelism, the exponential performance gains attained by leveraging hardware featuring an increasing core number per CPU continued, but as soon as several multicore CPUs were combined to one node, and multiple nodes added their performance to a cluster, also communication became a relevant factor that limited the applications' performance. An additional restriction that became obvious was the problem of energy consumption as gathering thousands of CPUs into one cluster also adds their power draft.

To address these issues, around 2006, hardware developers started looking back into less complex cores. Although these often have the drawback of supporting only a reduced instruction set and featuring a less sophisticated cache hierarchy, they exhibit a clear advantage in power consumption so that a large number can be aggregated into one device. *Graphics Processing Units* (GPUs), even though they have a different historical background, are one example where the gathering of less-complex cores into one device is taken to extremes. But while GPUs provide excellent performance for highly parallel applications (also with respect to their power draft [KBD10]), they are not appropriate to tackle complex tasks. Therefore, one often merges them into a system which is also equipped with complex or conventional cores, forming a *Heterogeneous Computing* platform. While these architectures usually combine high performance with adequate energy efficiency, the real challenge is left to the programmers, as leveraging the provided computational power is even more difficult: now the application has to be broken down into parallel tasks, and also each task has to be assigned to the suitable hardware device. Despite the difficulties in leveraging the computational power, it seems that the trend for an increasing level of heterogeneity will dominate the future in High-Performance Computing, enabling the continuation of the exponential growth in computing power and paving the way to Exascale computing [BBC+08].

1.1.2. Software Impact

In spite of all the elation about exponential growth in performance, the burden of leveraging the computational power remains with the software developers and the application designers, who are supposed to derive algorithms suitable for platforms that, in a near future, will typically consist of heterogeneous processors, featuring different instruction sets and able to execute certain tasks with varying performance. This may become even more difficult with the additional complexity of the memory models: While the uni-core era was based on a straight-forward memory hierarchy, the rise of multicore systems came along with the possibilities of a unified- and/or a distributed-memory address space [KBD10], both with its advantages and drawbacks.

These hardware shifts have significant impact on the software implementations. The largest problem arises from the fact that old legacy codes are not able to automatically take advantage of the new hardware technologies. Due to the growing gap in peak performance between single core and multi-core/many-core devices, the single-threaded programs tend to perform even worse on the emerging platforms [Luk12]. Furthermore, applications designed for clusters often neither utilize the full power potential of modern multi-core CPUs, due to the different synchronization mechanisms, nor are they able to leverage the computing power of accelerators, since most many-core platforms support explicit communication control. Thus, especially in scientific computing, many of the existing simulation algorithms have to be adapted, reprogrammed or even completely redesigned by changing the underlying numerical methods in order to achieve full utilization of the new hardware [BBC$^+$08, DBM$^+$11].

1.1.3. Computational and Technical Challenges to Overcome for Exascale Computing

Already the most recent hardware developments revealed that the obstacles in transforming theoretical computational power into simulation performance are sufficiently complex such that a new methodological approach is necessary when aiming for Exascale simulations. Particularly, the interaction between hardware, software and numerical methods requires that research specialists of the different areas including applied mathematics, computer science, hardware engineering and ultimately application scientists merge their competencies and interact as they pursue their own research agendas [ABC$^+$10, DBM$^+$11]. The necessity of collaboration stems from the characteristics of the technical and computational hurdles that have to be addressed. Many research reports identify three main challenges [ABC$^+$10, BBC$^+$08, Age10, DBM$^+$11]:

- *Exploiting massive parallelism.*
  Mathematical models, numerical methods and software implementations need new conceptual and programming paradigms to make effective use of unprecedented levels of concurrency. The scaling with respect to multiple millions up to billions of processing units requires the exploitation of all types of parallelism in the programs and the reduction of communication to almost nothing.

- *Coping with run-time errors.*
  An immediate consequence of the high component number of future hardware systems is that the frequency of hardware errors will increase significantly, while timely identification and error correction will become even more difficult. The ability to cope with hardware failure and a strong tolerance of the algorithms with respect to latencies is essential to maintain the resilience of simulation algorithms.

- *Reducing power requirements.*
  Based on current technology, scaling today's systems to exaflop level would imply

power consumption of hundreds of megawatts [ABC+10]. While strong efforts from the hardware manufacturers are required to reduce this resource demand, they come along with the necessity of shifting the focus from pure runtime performance to more energy-aware efforts to optimize application algorithms. This demands for adapting the implementations to the evolving hardware as well as leveraging the power-saving mechanisms provided by the systems.

1.2. Goal and Thesis Contributions

Implementation of efficient and scalable numerical methods suitable for the next generation of High Performance Computers is an interdisciplinary task which requires ample knowledge of applied mathematics and computer science. Especially, the design and implementation of linear system solvers featuring properties required for exascale computing is a nontrivial venture.

In this thesis this ambitious challenge is addressed by deriving block-asynchronous iterative methods suitable for hardware platforms accelerated by Graphics Processing Units, since these may belong to the most accepted hardware accelerators or coprocessors. The proposed methods combine scalability, fault-tolerance and energy efficiency, often identified as key properties for future Exascale implementations. Tackling this problem at the node-level can be considered as a first step towards the final goal of solving the problem at a much larger scale and, ultimately, at Exascale. In this line, it may be assumed that the asynchronous properties inherently allow for this problem reduction becoming potential candidates to be considered in future exascale simulation.

The main contributions of the thesis are:

- **Derivation of block-asynchronous iteration.**
  We design a block asynchronous iterative method able to solve linear systems of equations and suitable for GPU-accelerated heterogeneous hardware platforms. In our experiments we investigate its convergence and performance as well as the non-deterministic behavior and the tolerance to hardware failure. We furthermore optimize the algorithm to deal with sparse systems and enable the implementation of multi-GPU usage.
- **Problem-aware block-asynchronous iterative methods.**
  Block-asynchronous iterative methods can be adapted to a given problem by using weights or, in case of a discretized partial differential equation, by accounting for the discretization scheme. We provide convergence theorems for different weighting techniques and analyze in experiments the convergence and performance benefits when optimizing the algorithms to a specific application.
- **Block-asynchronous mixed precision iterative refinement.**
  We employ block-asynchronous iteration as an error correction solver in a mixed precision iterative refinement framework and investigate its convergence and performance for different GPU architectures. By handling also the residual computations of the iterative refinement loop asynchronously, we design a block-asynchronous iterative refinement. For this algorithm we study its theoretical convergence as well as provide experimental results analyzing the trade off between synchronous and asynchronous residual computation.
- **Block-asynchronous multigrid smoothers**
  Since component wise relaxation methods typically provide important contribution as smoother in multigrid methods, we investigate the potential of replacing the traditionally applied synchronized algorithms by the block-asynchronous iteration.

- **Energy considerations.**
  Recent hardware typically features a variety of energy-saving mechanisms. The algorithms' energy footprint depend on the potential of leveraging these techniques. For different iterative solvers the optimization potential is evaluated and experimental results are presented. Furthermore, we put particular focus on analyzing the energy efficiency of block-asynchronous iteration.

While these contributions are mainly presented in Chapter 4 and 5, some of the theoretical and experimental results have, at least partially, already been published in conference proceedings (see Table 1.1).

1.3. Thesis Outline

Chapter 2 provides some theoretical background about iterative methods traditionally employed when solving linear systems of equations. This includes iterative refinement methods, relaxation methods, the iterative approach to Krylov subspace solvers as well as multigrid methods. While these methods all share the common principle of synchronizations in the computing process of the sequence of approximations, in Chapter 3 we review some existing theory about component wise asynchronous iteration methods. This background is useful in the derivation of the GPU-adapted block-asynchronous iteration in Chapter 4. We begin this chapter with some fundamental principles in the context of general purpose computing on graphics processing units, motivating for the algorithm design. In the following sections we examine different aspects of the method. Particularly, we investigate the non-deterministic behavior when implementing block-asynchronous iteration on hardware platforms accelerated by graphics processing units and the tolerance to hardware failure. We then analyze how block-asynchronous iteration can efficiently be applied to dense or sparse linear systems using either one GPU or a cluster of GPUs. After deriving some theory on how the algorithm can be enhanced by the use of weights and proving convergence for the different weighting techniques, we analyze how block-asynchronous iteration can efficiently be adapted to discretized partial differential equations. We also target the question whether it is beneficial to employ block-asynchronous iteration as smoother in multigrid solvers or as error correction solver in mixed precision iterative refinement. Associated to the latter idea we derive some theory on block-asynchronous iterative refinement, where also the residual computation is handled synchronization-free. All these theoretical studies are supplemented by numerical experiments analyzing convergence behavior and runtime performance of the different implementations. Finally we conclude this chapter, which may be seen as gist of the thesis, by using block-asynchronous iteration in the solution process of a nonlinear instationary partial differential equation arising in the simulation of pattern formation in mathematical biology. Later on, we address in Chapter 5 the issue of energy-efficient simulation algorithms. This includes not only the introduction of different power and energy saving mechanisms, but moreover experimental results on traditional methods and block-asynchronous iteration obtained from a sophisticated power measurement setup. In Chapter 6 we summarize the contributions and findings of this work and provide an outlook how the derived methods can efficiently be applied in scientific simulation algorithms, and which challenges should be addressed in the further research on this topic.

Section	Description	New contribution
4.1	review of GPU-Computing	-
4.2	BAI for GPUs	[ATDH12a]
4.3	analysis of non-deterministic behavior of BAI	[ATDH12a]
4.4	experiments on BAI	[ATDH12a]
4.5	fault-tolerance properties of BAI	unpublished
4.6	BAI for multi-GPU sytems	unpublished
4.7	experiments on BAI for multi-GPU sytems	unpublished
4.8.1	review of CSR-format	-
4.8.2	modified CSR for BAI	unpublished
4.9	experiments on modified CSR for BAI	unpublished
4.10.1	review of weighting techniques	-
4.10.2	weighted BAI	[ATDH12b]
4.10.2.1	ω-weights for BAI	[ATDH12b]
4.10.2.2	convergence of ω-weighted BAI	unpublished
4.10.2.3	ℓ_1-weights for BAI	[ATDH12b]
4.10.2.4	convergence of ℓ_1-weighted BAI	unpublished
4.10.3	experiments on ω-/ℓ_1-weighted BAI	[ATDH12b]
4.10.4	θ_i-weighted BAI	unpublished
4.10.5	experiments on θ_i-weighted BAI	unpublished
4.10.6	PDE-aware BAI	unpublished
4.10.7	PDE-aware BAI adapted to Helmholtz	unpublished
4.10.8	experiments on PDE-aware BAI adapted to Helmholtz	unpublished
4.11.1	review of multigrid smoothers	-
4.11.2	BAI in multigrid methods	[ATG+12]
4.12	BAI in mixed precision iterative refinement	[ALDH12]
4.13	asynchronous iterative refinement	unpublished
4.13.1	theoretical aspects of asynchronous iterative refinement	unpublished
4.13.2	theoretical aspects of asynchronous mixed precis. iter. ref.	unpublished
4.13.3	theoretical aspects of block-asynchronous iterative refinement	unpublished
4.13.4	experiments on block-asynchronous iterative refinement	unpublished
4.14	review of non-linear PDEs	-
4-14.1	experiments of BAI applied to a nonlinear PDE	unpublished
4.14.2	experiments on BAI applied to pattern formation problem	unpublished
5.1	review of energy-aware computing	-
5.2.1	energy measurement setup at HPCA	-
5.2.2	energy measurement setup EMCL	[ABC+]
5.3.1	review of accelerators energy-aware computing	-
5.3.2	review of dynamic voltage and frequency scaling (DVFS)	-
5.3.3	introduction of idle-wait technique	[AHA+11]
5.4.1	experiments on energy-savings by accelerator computing	[AHA+11]
5.4.2	experiments on energy-savings by DVFS	[AHA+11]
5.4.3	experiments on energy-savings by idle-wait	[AHA+11]
5.4.4	experiments on energy-efficiency of BAI	unpublished

Table 1.1.: Overview about own contributions in Chapter 4 and 5 and related publications. The term BAI refers to block-asynchronous iteration.

2. Classical Iterative Methods

This chapter is dedicated to procure an overview about the classical iteration methods used for finding an approximation for the solution of systems of linear equations of the form

$$Ax = b \quad \text{with } A \in \mathbb{R}^{n\times n} \text{ and } x,\ b \in \mathbb{R}^n. \tag{2.1}$$

The purpose is not limited to introducing the reference methods for numerical experiments on the asynchronous schemes we will investigate in Chapter 4, but moreover to provide the mathematical background necessary to derive the hardware-aware numerics in the following chapters.
First we describe in Section 2.2 the iterative refinement method which is modified in Section 2.3 by adapting it to modern hardware systems. We then introduce in Section 2.4 the classical relaxation methods based on matrix splitting like Jacobi and Gauss-Seidel. The underlying idea of component wise updates will also be used in the asynchronous and block-asynchronous methods we derive and investigate in Chapter 3 and 4. For the Conjugate Gradient method and GMRES we derive the algorithms and show some complexity estimations based on convergence theory (see Section 2.5.3, 2.5.4). Finally, we introduce multigrid methods. While these usually base on the properties of the finite element discretization of a partial differential equation, they may be formulated as iterative method too.

2.1. Iterative Methods

In numerical mathematics, iterative solvers are methods that compute an approximation of the solution to a given problem, starting from an initial guess. In each iteration step of the solver, the solution approximation is improved until a preset accuracy threshold is reached, and the algorithm is stopped (see e.g. [Ran08, Saa03, Bou01]). Starting from the initial guess x^0, iterative methods usually generate a sequence of solution approximations $x^0, x^1, x^2, \dots x^k \dots$ that, in the convergent case, approaches the exact solution in its limit. While iterative methods are usually the only choice for nonlinear equations, also in the case of large linear systems they are often preferred to direct solvers, since they are able to compute a sufficiently accurate approximation of the solution with comparably low computational effort [Saa03].

For a linear system $Ax = b$, iterative solvers base on an iteration algorithm that is stopped when a certain stopping criterion is fulfilled, and an initial solution guess. As the absolute error of a solution approximation is unknown to the point where the exact solution is computed, one normally uses a threshold for the residual error term $r^k = b - Ax^k$ as stopping criterion. Especially the threshold is often chosen relative to the initial error, such that a typical stopping criterion takes the form

$$\frac{\| r^k \|_2}{\| r^0 \|_2} \leq \varepsilon. \tag{2.2}$$

An interesting question is, how the absolute error and the residual error depend on each other. As specified in the notation, we denote $e^k = x^* - x^k$ the absolute error in the kth step, and $r^k = b - Ax^k$ the residual or residual error of the ith step. With some elementary computations one can obtain

$$\begin{aligned}
& x^* = A^{-1}b \\
\Leftrightarrow \quad & x^* - x^k = A^{-1}b - x^k \\
\Leftrightarrow \quad & e^k = A^{-1}b - A^{-1}Ax^k \\
\Leftrightarrow \quad & e^k = A^{-1}\underbrace{(b - Ax^k)}_{=r^k} \\
\Leftrightarrow \quad & e^k = A^{-1}r^k,
\end{aligned} \tag{2.3}$$

and especially

$$\| e^k \|_2 = \| A^{-1}r^k \|_2 \leq \| A^{-1} \|_2 \| r^k \|_2 \tag{2.4}$$

$$\| r^k \|_2 = \| Ae^k \|_2 \leq \| A \|_2 \| e^k \|_2 \tag{2.5}$$

which provides the direct dependency between r^k and e^k.
For the further theory, we introduce the condition number of a matrix A with respect to a matrix norm $\| \cdot \|_p$ as [Gre87]

$$\kappa_p(A) := \| A \|_p \cdot \| A^{-1} \|_p . \tag{2.6}$$

With (2.4) and (2.5) we can derive

$$\frac{1}{\kappa_p(A)} \frac{\| r^k \|_p}{\| b \|_p} \leq \frac{\| e^k \|_p}{\| x^* \|_p} \leq \kappa_p(A) \frac{\| r^k \|_p}{\| b \|_p} \tag{2.7}$$

which shows that the condition number of the matrix A has significant influence on the dependency between error and residual [Gre87].

It is important to point out the difference between the absolute error term e^k and residual error term r^i. In the analysis of iterative solvers one often has to deal with an discrepancy concerning the different stopping criteria in the theoretical analysis and the numerical tests on implementations. On the theoretical side, the whole convergence analysis is usually based on the error term e^k and an absolute error stopping criterion. Implementing an iterative solver, the error is usually unknown, as the exact solution is unknown. Hence, one usually chooses a residual error stopping criterion. This inconsistency could be avoided by using the equation (2.3) connecting the residual error to the absolute error. But as the transformation of the absolute error into the residual error is at least difficult, or even impossible due to the unknown matrix A^{-1}, we want to limit our information about the dependency to the fact, that for sufficiently well-conditioned systems, a small absolute error usually implies a small residual and vice versa (see equation (2.7)).

2.2. Iterative Refinement

Although *iterative refinement methods* have been known for long time, they have enjoyed a revival with the rise of computer systems in the middle of the last century [GST07]. The core idea is to use the residual of a computed solution as the right-hand side to solve a correction equation [GST07]. The algorithm then updates the solution approximation in every iteration by adding an error correction term computed by an *error correction solver*. We want to stress that this error correction solver can be chosen independently: direct solvers as well as another iterative method are possible options. This implies the possibility of cascading iterative refinement methods.

Denoting the solution update with $c^k := A^{-1}r^k$, the algorithm can be formulated like in Algorithm 1.

Algorithm 1 Iterative Refinement Methods.

1: initial guess as starting vector: x^0
2: compute initial residual: $r^0 = b - Ax^0$
3: **while** ($\| Ax^k - b \|_2 > \varepsilon \| r^0 \|_2$) **do** {outer iteration}
4: $r^k = b - Ax^k$
5: solve $Ac^k = r^k$ {inner iterations when using an iterative method}
6: update solution $x^{k+1} = x^k + c^k$
7: k=k+1
8: **end while**

In the beginning of the method, an initial approximation x^0 is guessed. In each iteration, the inner correction solver searches for a c^k, such that $Ac^k = r^k$ with r^k being the residual of the solution approximation x^k. Then, the approximation of the solution x^k is updated to $x^{k+1} = x^k + c^k$.

2.2.1. Convergence of Iterative Refinement Methods

If we denote the residual in the ith step of the solution process of (2.1) as

$$r^k = b - Ax^k$$

we can analyze the improvement gain by performing one iteration loop of the iterative refinement method.
Applying an error correction solver to the equation $Ac^k = r^k$ which generates a solution approximation with a relative residual error of at most $\varepsilon_{\text{inner}} \| r^k \|_2$, we get an error correction term c^k, fulfilling

$$r^k - Ac^k = d^k, \tag{2.8}$$

where d^k is the residual of the correction solver with the property $\| d^k \|_2 \leq \varepsilon_{\text{inner}} \| r^k \|_2$. In the case of an iterative error correction solver, the threshold $\varepsilon_{\text{inner}} \| r^k \|_2$ can be chosen as residual stopping criterion, while in the case of direct correction solvers, $\varepsilon_{\text{inner}} \| r^k \|_2$ is an upper bound for the residual of the solution approximation [AHR10].

Updating the solution $x^{k+1} = x^k + c^k$ we obtain for the new residual error term

$$\begin{aligned} \| r^{k+1} \|_2 &= \| b - Ax^{k+1} \|_2 \\ &= \| b - A(x^k + c^k) \|_2 \\ &= \| \underbrace{b - Ax^k}_{=r^k} \underbrace{- Ac^k}_{=d^k - r^k} \|_2 \\ &= \| d^k \|_2 \leq \varepsilon_{\text{inner}} \| r^k \|_2 . \end{aligned}$$

Hence, the accuracy improvement obtained by performing one outer iteration loop equals the accuracy of the inner error correction solver. Using this fact, one can prove by induction that after i iteration loops the residual r^k fulfills

$$\| r^k \|_2 \leq \varepsilon_{\text{inner}}^k \| r^0 \|_2 . \tag{2.9}$$

To obtain the number of iterations i necessary to get a residual term $\| r^k \|_2 \leq \varepsilon \| r^0 \|_2$, we use the properties of the logarithm and the estimation

$$\begin{aligned} \| r^k \|_2 &\leq \varepsilon \| r^0 \|_2 \\ \varepsilon_{\text{inner}}^k \| r^0 \|_2 &\leq \varepsilon \| r^0 \|_2 \\ \varepsilon_{\text{inner}}^k &\leq \varepsilon \\ i &\geq \frac{\log \varepsilon}{\log \varepsilon_{\text{inner}}}. \end{aligned}$$

Since *outer_iterations* has to be an integer, the Gaussian ceiling function can be applied to obtain

$$outer\_iterations = \left\lceil \frac{\log(\varepsilon)}{\log(\varepsilon_{\text{inner}})} \right\rceil \tag{2.10}$$

for the number of outer iterations necessary to get a residual below the threshold $\| r^k \|_2 \leq \varepsilon \| r^0 \|_2$ [AHR10]. Note that this estimation does not provide information about the total computational cost, since the number of outer iterations does not reflect the computational work of the error correction solver. Especially, using an iterative error correction solver, the total number of iterations is usually considerably higher than the number of outer iterations. The performance in the end depends on the trade-off between stopping criterion of the inner error correction solver and the number of outer iterations.

2.3. Mixed Precision Iterative Refinement

The iterative refinement method described in the previous section is not only very flexible in terms of choosing the error correction solver, but also with respect to the floating point formats used in the different parts. The underlying idea of *mixed precision iterative refinement methods* is to leverage this flexibility by computing the error correction term in lower precision than working precision (see e.g. [GST07, BBD+09, BDL+07] and the references therein).

In this setup one regards the inner correction solver as a black box, computing a solution update in lower precision. Using the term high precision (X^{high}) to denote the precision format that is necessary to display the accuracy of the final solution and low precision (X^{low}) for values in the lower precision format, we can formulate Algorithm 2 (see Figure 2.1) which, especially when targeting hardware platforms that can compute at higher speed when using less complex floating point formats, may be beneficial to the overall runtime performance [AHR11b].

Algorithm 2 Mixed precision approach to iterative refinement.

1: convert system matrix to low precision $A^{low} = A^{high}$
2: set initial values: $x^{high} = 0$
3: **while** $\| b^{high} - A^{high}x^{high} \|_2 \geq \varepsilon \| r^0 \|_2$ **do**
4: compute error in high precision $r^{high} = b^{high} - A^{high}x^{high}$
5: convert residual to low precision for inner solver $r^{low} = r^{high}$
6: solve the correction equation in lower precision $A^{low}c^{low} = r^{low}$
7: convert error correction term to high precision $c^{high} = c^{low}$
8: update outer solution $x^{high} = x^{high} + c^{low}$
9: **end while**
10: give the solution x^{high} in high precision

Since especially the conversion of the matrix A into the low precision format is expensive, implementations of mixed precision iterative refinement usually store it in both precision formats. Furthermore, in the case of using hybrid hardware, A should be stored in the local memory of the hardware components in the respectively used format [BBD+09].

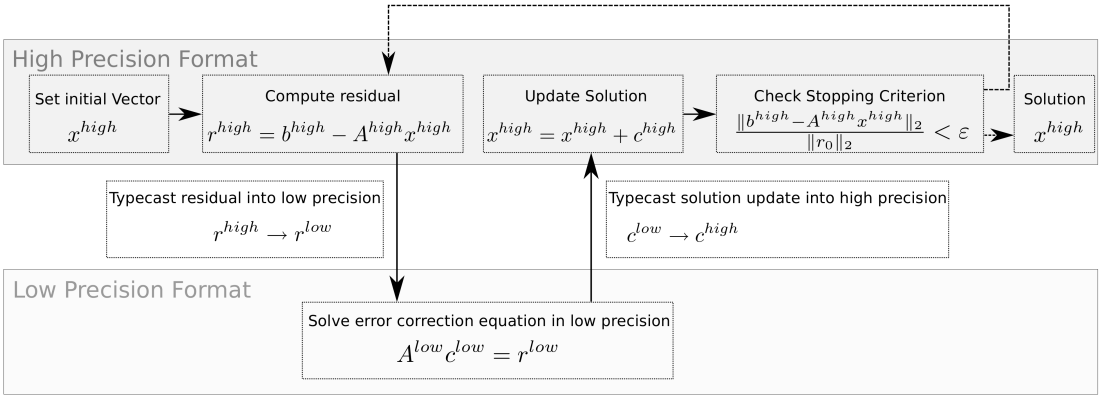


Figure 2.1.: Visualizing the mixed precision approach to an iterative refinement solver.

Like in the case of using one format, the computation of the correction loop $A^{low}c^{low} = r^{low}$ is independent and can be handled by a direct solver, or again by an iterative method. This implies that it is possible to cascade multiple iterative refinement solvers using successively decreasing precision. Theoretically, any floating point format can be chosen, but it is mostly reasonable to employ the widespread used IEEE 754 (see Appendix A.1), as it is supported by most hardware.

When comparing the algorithm of an iterative refinement solver using a certain iterative solver as error correction solver to the plain iterative solver in high precision, we realize, that the iterative refinement method has more computations to execute due to the additional residual computation, solution updates and typecasts. For this reason, one interesting question is, in which cases the mixed precision iterative refinement method outperforms the plain solver in high precision. Some aspects on this issue are presented in [AHR10].

2.3.1. Convergence Analysis of Mixed Precision Approaches

When discussing the convergence of the iterative refinement method in Section 2.2.1, we derived a model for the number of outer iterations that are necessary to obtain a residual error below a certain residual threshold ($\varepsilon \parallel r^0 \parallel$): Having a relative residual stopping criterion ε_{inner} for the iterative error correction solver, we need to perform

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

iterations to obtain an approximation x^k which fulfills

$$\parallel r^k \parallel_2 = \parallel b - Ax^k \parallel_2 \leq \varepsilon \parallel b - Ax^0 \parallel_2 = \varepsilon \parallel r^0 \parallel_2 .$$

If we now use the iterative refinement technique in mixed precision, we have to modify this convergence analysis to account for the rounding effects triggered by the floating point arithmetic. In fact, two phenomena may occur that require additional outer iterations[AHR10]:

1. Independently of the type of the inner error correction solver, the low precision representation of the matrix A and the residual r^k contain representation errors due to the floating point arithmetic. These rounding errors imply that the error correction solver is applied to a perturbed system $(A+\delta A)c^k = r^k + \delta r^k$. Due to this fact, the solution update c^k may give less improvement to the outer solution than expected.
2. When using iterative error correction solvers that compute the residual within the iteration process (e.g. Krylov subspace methods) the iteratively computed residuals may differ from the actual residuals due to accumulated rounding errors. This potentially triggers early or late breakdowns of the error correction solver.

It should also be mentioned, that convergence of the mixed precision iterative refinement algorithm is only achieved for cases where the iterative error correction solver converges in the respectively used floating point format. If we denote the total number of additional outer iterations, induced by the rounding errors and the early breakdowns when using an iterative error correction solver, with g, we can obtain

$$outer\_iterations_{total} = \left\lceil \frac{\log \varepsilon}{\log \varepsilon_{inner}} \right\rceil + g \tag{2.11}$$

for the total number of outer iterations necessary to get an approximation that fulfills $\parallel r^k \parallel_2 \leq \varepsilon \parallel r^0 \parallel_2$. At this point we want to mention, that g in fact does not only depend on the type of the error correction solver and the used floating point formats, but also on the rounding schemes used for the conversion in-between and the properties of the linear problem including the matrix structure.

2.4. Component Wise Relaxation Methods

Classical relaxation algorithms are methods for finding an approximate solution for the linear system of equations (2.1)

$$Ax = b \quad \text{with } A \in \mathbb{R}^{n \times n} \text{ and } x,\ b \in \mathbb{R}^n$$

that are based on relaxation of the components, and in a convergent iteration process they generate a sequence of solution approximations with increasing accuracy. While the

convergence of relaxation methods is rarely guaranteed for all matrices, there exists a comprehensive convergence theory for cases where the coefficient matrix derives from the finite difference discretization of elliptic partial differential equations [Saa03]. Given an initial solution approximation, every iteration step modifies one or multiple components, where the different components are updated in a certain order. Although these techniques are today rarely directly applied to a linear equation problem, they often provide important contribution to complex solvers, e.g. as preconditioner or smoother in a multigrid framework (see Section 2.6).

The *Banach fixed point theorem* [Ban22, SK06] guarantees the existence and uniqueness of fixed points for self-mapping contraction operators. The idea of component wise relaxation is to decompose the matrix A to derive a contraction mapping of the form

$$\varphi(x) = B \cdot x + d, \tag{2.12}$$

where the spectral radius $\rho(B)$ is smaller than one. Then, the iteration method is defined by

$$x^{k+1} = B \cdot x^k + d$$

where B is usually denoted the *iteration matrix*. The various component wise relaxation methods then basically differ in the choice of B and d. A relaxation method can be derived by considering an invertible matrix M and decomposing

$$A = M + (A - M). \tag{2.13}$$

Then,

$$\begin{aligned} x^{k+1} &= M^{-1}(b - (A - M))x^k \\ &= M^{-1}(M - A)x^k + M^{-1}b \\ &= (I - M^{-1}A)x^k + M^{-1}b. \end{aligned} \tag{2.14}$$

Hence, we get for any matrix splitting an iteration method with $B = (I - M^{-1}A)$ and $d = M^{-1}b$. Therefore, a relaxation scheme is equivalent to a fixed point iteration applied to the preconditioned linear equation system [Saa03]

$$M^{-1}Ax = M^{-1}b.$$

Every relaxation method can also be written in component wise form:

$$x_i^{k+1} = \sum_{j=1}^{n} b_{i,j} \cdot x_j^k + d_i, \tag{2.15}$$

where $B = (b_{i,j})$ is the iteration matrix.

The classical relaxation methods can be derived by using different matrix splittings. The coefficient matrix A in (2.1) can be decomposed into a sum of matrices $A = A_1 + A_2 + A_3 \cdots + A_n$. One very popular decomposition is

$$A = L + D + U \tag{2.16}$$

where D contains the diagonal entries of A and L respectively U contain the lower and upper triangular part of A (see Figure 2.2). Using this decomposition, we may rewrite (2.1) into

$$(L + D + U)x = b.$$

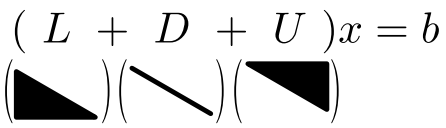


Figure 2.2.: Decomposition of A in diagonal, upper and lower triangular part.

This decomposition serves as basis for the Jacobi, Gauss-Seidel and backward Gauss-Seidel methods we derive in the following sections. At this point it should be mentioned, that the decomposition in diagonal, upper and lower triangular part is often also used in blockwise fashion, see Section 2.4.4.

It seems reasonable to introduce some notations that simplify the comparison of the different methods. We denote with

- **local iterations** the individual component updates,
- **global iterations** an ordered sequence of local iterations (component updates).

Note, that from one iterate x^k we obtain the next global iterate x^{k+1} by updating every component exactly once, while the order in which the different components are updated is in general preset.

2.4.1. Jacobi Method

The Jacobi[1] method derives by using the matrix decomposition (2.16) and transforming the system of linear equations (2.1) into

$$\begin{aligned} & Ax = b \\ \Leftrightarrow \quad & (L+D+U)x = b \\ \Leftrightarrow \quad & Dx = b - (L+U)x \\ \Leftrightarrow \quad & x = \underbrace{\left(I - D^{-1}A\right)}_{\text{iteration matrix } B} x + \underbrace{D^{-1}b}_{\text{additive component } d}. \end{aligned} \tag{2.17}$$

Hence, we choose $M = D$ in (2.13) and the iteration matrix becomes $B = D^{-1}(-L-U) = I - D^{-1}A$. The obtained Jacobi iteration process

$$x^{k+1} = D^{-1}b - D^{-1}(L+U)x^k \tag{2.18}$$

can also be rewritten in the component wise form

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=0, j\neq i}^{n} a_{i,j} x_j^k \right) = \sum_{j=0}^{n} b_{i,j} x_j^k + d_i \tag{2.19}$$

where

$$B = (b_{i,j}) = \begin{pmatrix} 0 & -\frac{a_{12}}{a_{11}} & -\frac{a_{13}}{a_{11}} & \cdots & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & \ddots & \vdots & \\ -\frac{a_{31}}{a_{33}} & \ddots & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \\ -\frac{a_{n1}}{a_{nn}} & -\frac{a_{n2}}{a_{nn}} & -\frac{a_{n3}}{a_{nn}} & \cdots & 0 \end{pmatrix} \tag{2.20}$$

and $d_i = \frac{b_i}{a_{ii}}$.

[1] Carl Gustav Jacob Jacobi (⋆1804, †1851)

We observe that the individual components can be updated in parallel for one global iteration. According to the Banach fixed point theorem, it provides a sequence of solution approximations with increasing accuracy when the spectral radius of the iteration matrix B is less than one (i.e., $\rho(I - D^{-1}A) < 1$) [Bag95].

2.4.2. Gauss-Seidel Method

Like the Jacobi method, the Gauss[2]-Seidel[3] (G.-S.) method derives from the matrix splitting (2.16):

$$\begin{aligned} & Ax = b \\ \Leftrightarrow \quad & (L + D + U)x = b \\ \Leftrightarrow \quad & (D + L)x = b - Ux \\ \Leftrightarrow \quad & x = (D + L)^{-1}b - (D + L)^{-1}Ux. \end{aligned} \tag{2.21}$$

Hence, using $M = D + L$ in (2.13) the Gauss-Seidel iteration method becomes

$$x^{k+1} = (D + L)^{-1}(b - Ux^k). \tag{2.22}$$

Similarly to Jacobi, Gauss-Seidel iteration corrects the i-th component of the solution approximation in the i-th local iteration, but instead of only old values, it uses in the i-th local iteration the already updated values for the components x_j for $j < i$. Therefore, the update order for the distinct components is crucial: the scheduling for the local iterations forming one global iteration has to be adhered. This can also easily be deduced from the component wise algorithm description

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{k+1} - \sum_{j=i+1}^{n} a_{i,j} x_j^k \right). \tag{2.23}$$

While Gauss-Seidel updates the components in the order $1, 2 \ldots n$, we can derive a backward Gauss-Seidel by the update order $n, n-1 \ldots 1$ with the update algorithm

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{k} - \sum_{j=i+1}^{n} a_{i,j} x_j^{k+1} \right). \tag{2.24}$$

This corresponds to the matrix splitting (choose $M = D + U$ in (2.13))

$$\begin{aligned} & Ax = b \\ \Leftrightarrow \quad & (L + D + U)x = b \\ \Leftrightarrow \quad & (D + U)x = b - Lx \\ \Leftrightarrow \quad & x = (D + U)^{-1}b - (D + U)^{-1}Lx. \end{aligned} \tag{2.25}$$

and the algorithm

$$x^{k+1} = (D + U)^{-1}(b - Lx^k). \tag{2.26}$$

A symmetric Gauss-Seidel consists of a forward sweep followed by a backward sweep [Saa03].

[2] Carl Friedrich Gauß(⋆1777, †1855)
[3] Philipp Ludwig von Seidel (⋆1821, †1896)

2.4.3. SOR Method

The Jacobi and the Gauss-Seidel iterations are both of the form

$$x^{k+1} = M^{-1}\left(M - A\right)x^k + M^{-1}b,$$

where $M = D$ for Jacobi, $M = D + L$ for Gauss-Seidel and $M = D + U$ for backward Gauss Seidel. A very similar method can be derived, by using a weighted splitting [Var10, BBC+94]:

$$\begin{aligned} & \omega Ax = \omega b \qquad (2.27)\\ \Leftrightarrow \quad & (D + \omega L)x + (\omega U - (1-\omega)D)x = \omega b \\ \Leftrightarrow \quad & x = \omega(D + \omega L)^{-1}b - (D + \omega L)^{-1}(\omega U - (1-\omega)D)x. \end{aligned}$$

The obtained algorithm reads

$$x^{k+1} = (D + \omega L)^{-1}(\omega b - (\omega U - (1-\omega)D)x^k). \qquad (2.28)$$

and is usually denoted as *Successive Over Relaxation* (SOR). An interesting connection is, that this method can also be derived by weighting the Gauss-Seidel algorithm [Saa03]:

$$x_{SOR}^{k+1} = \omega x_{G.-S.}^{k+1} + (1-\omega)x_{G.-S.}^{k}.$$

Similar to the backward Gauss-Seidel it is possible to define backward SOR. Finally, a symmetric SOR (SSOR) step consists of an SOR step followed by a backward SOR step [Saa03].

2.4.4. Block Relaxation Schemes and Two-Stage Iteration Methods

As already indicated, all component wise iteration methods can also be extended to a block version. In this case, the matrix, the iteration vector and the right hand side of the system (2.1)

$$Ax = b$$

of dimension n is decomposed into the block system

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,q} \\ A_{2,1} & A_{2,2} & \dots & A_{2,q} \\ \vdots & \vdots & \ddots & \vdots \\ A_{q,1} & A_{q,2} & \dots & A_{q,q} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_q \end{pmatrix}. \qquad (2.29)$$

Note that while the decomposition into the $q \times q$ blocks is not necessarily uniform, it may happen that some blocks are larger than others, while the diagonal blocks A_{ii} are required to be square of order n_i, $i = 1, \dots q$, and $\sum_{i=1}^{q} n_i = n$ [Fro94].

Block versions of component wise iterative methods, also called *block relaxation schemes*, are not only very interesting when solving large linear systems on parallel computers [Var10, BFG+], but also very popular when implementing parallel preconditioners for multigrid methods [BFKMY11]. The idea is to use a splitting $A = M + (A - M)$ where M is the block diagonal, denoted $M = diag(M_i)$, with the nonsingular blocks $M_i = A_{i,i}$ of order n_i, $i = 1 \dots q$. Using this decomposition, a block iterative method can then be derived as the solution of the respective block equations

$$M_i x_i = ((M - A)x + b)_i \qquad (2.30)$$

and the global update. As, for different update orders, the Jacobi, Gauss-Seidel and SOR method can be derived with the difference of now taking matrix blocks and subvectors

instead of scalar components. An interesting case occurs if the system of linear equations is derived from the finite element discretization of an elliptic partial differential equation on a rectangular domain. Then, the block size can be chosen according to the discretization mesh. Adapting the block size to one line of the mesh, one obtains a so-called *line relaxation technique* [Saa03].

For example, the blockwise Jacobi iteration can be formulated:

Algorithm 3 Block Jacobi Algorithm [Fro94]

initial solution approximation x^0
while ($\| b - Ax^k \|_2 \geq \varepsilon \| r^0 \|_2$) **do**
  for ($i = 1; i < q; i++$) **do**
    solve $M_i x_i^{k+1} = ((M - A)x^k + b)_i$
  end for
end while

The solution process of the sub-equations is independent from the global iteration, and it is possible to handle it by different processors. Since the exact solution using a direct solver is usually very expensive, one often prefers to apply iterative methods also to the sub-problems. In this case the resulting cascaded iterative solvers are often denoted as *two-stage iterative methods* [Saa03]. The advantage of these two-stage iterative methods is the higher parallelism in the iteration process, which stems from the fact that, depending on the solver chosen for the sub-problems, all components aggregated in one subvector can be updated at the same time [Saa03].

2.4.5. Convergence of Relaxation methods

All the derived methods can be written as a sequence of iterates of the form (2.15)

$$x^{k+1} = Bx^k + d,$$

with iteration matrix B. We now want to focus on the convergence properties of this algorithm, especially we are interested in whether the iteration process converges, how fast the convergence is, and whether the limit is the solution of the linear equation problem (2.1). If the iteration (2.15) converges, its limit x satisfies for the general matrix splitting (2.13)

$$\begin{aligned} x &= Bx + d \\ &= M^{-1}(M - A)x + M^{-1}b \\ &= (I - M^{-1}A)x + M^{-1}b \\ &= x - M^{-1}Ax + M^{-1}b. \end{aligned}$$

Hence, it obviously is a solution of $Ax = b$. The question whether the iteration converges can be answered by the following theorem:

Theorem 2.4.1. *[Saa03] Let B be a square matrix such that the spectral radius $\rho(B) < 1$. Then, $I - B$ is nonsingular and the iteration (2.15)*

$$x^{k+1} = Bx^k + d,$$

converges for any d and x^0. Conversely, if the iteration (2.15) converges for any d and x_0, then $\rho(B) < 1$.

While it is usually very difficult to determine the spectral radius of the iteration matrix, an upper bound is sometimes sufficient such as $\rho(B) \leq \| B \|$ for any matrix norm $\| \cdot \|$. In [Bag95], Roberto Bagnara provides a convergence proof for the Jacobi and Gauss-Seidel iteration for the cases where A is strictly diagonally dominant or A is diagonally dominant and irreducible.

Theorem 2.4.2. *[Bag95] Let $A \in \mathbb{R}^{n\times n}$ be decomposed as in (2.16), and let $B_{JAC} = D^{-1}(D-A)$ and $B_{G.-S.} = (D+L)^{-1}U$, respectively. If furthermore A is strictly diagonally dominant or is diagonally dominant and irreducible, then*

1. *D and $D+L$ are invertible*
2. *$\rho(B_{JAC}) < 1$ and $\rho(B_{G.-S.}) < 1$.*

Proof. [Bag95]

1. If A is strictly diagonally dominant, we see that $a_{ii} \neq 0$ for all $i = 1 \dots n$, hence D and $D+L$ are nonsingular. If A is diagonally dominant and irreducible, A is nonsingular and has no zero row. On the other hand, if either D or $D+L$ is singular, then $d_{kk} = 0$ for some $k = 1 \dots n$. But then the row k has to be zero, a contradiction.
2. In order to show that the respective spectral radii are less than 1, we define the following matrices

$$A_{JAC} = \lambda D + L + U, \tag{2.31}$$
$$A_{G.-S.} = \lambda(D+L) + U, \tag{2.32}$$

and establish the following Lemma:

Lemma 2.4.3. *[Bag95] For each $\lambda \in \mathbb{R}$, with $|\lambda| \geq 1$, if A satisfies any of the following properties:*

a) A is strictly diagonally dominant (by rows or by columns);

b) A is diagonally dominant (by rows, or by columns);

c) A is irreducible;

then both A_{JAC} and $A_{G.-S.}$ satisfy the same properties.

Proof. [Bag95]

a) Let A be strictly diagonally dominant by rows (the proof for the other case is almost the same). By hypothesis, for each $i = 1, \dots n$,

$$|a_{i,j}| > \sum_{j\neq i} |a_{i,j}|.$$

If $|\lambda| \geq 1$ then, for each $i = 1 \dots n$,

$$\begin{aligned}
|\lambda a_{ii}| &= |\lambda||a_{ii}| \\
&> |\lambda| \sum_{j\neq i} |a_{i,j}| \\
&= |\lambda| \sum_{j=1}^{i-1} |a_{i,j}| + |\lambda| \sum_{j=i+1}^{n} |a_{i,j}| \\
&\geq |\lambda| \sum_{j=1}^{i-1} |a_{i,j}| + \sum_{j=i+1}^{n} |a_{i,j}| \quad \text{for the Gauss-Seidel iteration } (A_{G.-S.}), \\
&\geq \sum_{j=1}^{i-1} |a_{i,j}| + \sum_{j=i+1}^{n} |a_{i,j}| \quad \text{for the Jacobi iteration } (A_{JAC}).
\end{aligned}$$

b) Similar to a) with the difference, that the hypothesis ensures that strict inequality holds for

$$|\lambda a_{ii}| > |\lambda| \sum_{j \neq i} |a_{i,j}|$$

for at least one component.

c) Since the three matrices A, $A_{JAC}(\lambda)$, $A_{G.-S.}(\lambda)$, for $\lambda \neq 0$, have zero components in exactly the same locations, it follows that if a permutation matrix reduces one of these matrices, then it also reduces the other two matrices.

□

To conclude the Theorem 2.4.2 we note that all eigenvalues λ of B_{JAC} are the solutions of the equation

$$det(\lambda I - B_{JAC}) = 0. \tag{2.33}$$

Since

$$\begin{aligned} det(\lambda I - B_{JAC}) &= det(\lambda I + D^{-1}(L+U) \\ &= det(D^{-1}) det(\lambda D + L + U) \\ &= det(D^{-1}) det(A_{JAC}(\lambda)). \end{aligned}$$

So, for (2.33) to hold we need $det(A_{JAC}(\lambda)) = 0$, as we have already shown that D is nonsingular. But, since A_{JAC} is nonsingular for $|\lambda| \geq 1$, it follows that all of the eigenvalues of B_{JAC} are in the interior of the unit circle. Hence, $\rho(B_{JAC}) < 1$. Similarly, the eigenvalues of $B_{G.-S.}$ are all and only the solutions of the equation

$$det(\lambda I - B_{G.-S.}) = 0. \tag{2.34}$$

From

$$\begin{aligned} det(\lambda I + B_{G.-S.}) &= det(\lambda I + (D+L)^{-1}U) \\ &= det((D+L)^{-1}) det(\lambda(D+L) + U) \\ &= det((D+L)^{-1}) det(A_{G.-S.}(\lambda)) \end{aligned}$$

we obtain similar to the Jacobi case, that $\rho(-B_{G.-S.}) < 1$, and thus $\rho(B_{G.-S.}) < 1$

□

2.4.6. Implementational Aspects

The computational effort of one component update is dominated by the multiplication of the respective row of the iteration matrix with the former iteration vector. Hence, the computational effort of one global iteration including the update of every component is in the order of a matrix vector multiplication. The number of iterations necessary to achieve convergence usually varies for the different splittings. For example, Gauss-Seidel in many cases converges about twice as fast as Jacobi [QSS00], but at the same time, the parallelization is limited to a stage where two components can never be updated at the same time [Saa03]. This is a strong drawback when using highly parallel computing systems. For blockwise Gauss-Seidel, the components in one subvector can be updated in parallel, but if these local component updates are conducted in the Jacobi-wise fashion, the convergence rate suffers again. In terms of memory, Gauss-Seidel is more economical, since the new approximation can be overwritten over the same vector [Saa03].

2.5. Krylov Subspace Methods

Another important group of iterative solvers are the *Krylov subspace methods* [Saa03]. Krylov subspaces of dimension $r < n$ are generated by an $n \times n$ matrix A and a vector b of dimension n as the linear subspace spanned by the images of b under the first r powers of A, that is

$$\mathcal{K}_r(A, b) = \text{span}\{b, Ab, A^2b, \ldots, A^{r-1}b\}.$$

They were named after the Russian mathematician Alexei Krylov[4], and have different important properties, that are useful in the convergence analysis of the iterative solvers based on these subspaces [CK01].
For solving a linear system $Ax = b$ of dimension n, Krylov methods use a starting vector x^0 and the initial residual $r^0 = Ax^0 - b$ to generate a sequence of approximations

$$x^0 \to x^1 \to \cdots \to x^m.$$

In the absence of rounding errors the exact solution to the equation is reached after n steps at latest [Saa03, Bou01]. Every step of the algorithm contains a matrix-vector multiplication that is essential to generate the next vector of the Krylov subspace. Since rounding errors usually occur when working with floating point numbers, and the approximation x^m for $m << n$ often already provides a good approximation of the exact result, Krylov subspace methods can also be considered as iterative solvers. Especially in the case of large dimensions, the Krylov methods as iterative solvers are often preferred to direct solvers [Saa03].

A first method of this kind is the *CG Algorithm* (see Section 2.5.3) which was invented by Hestenes[5] and Stiefel[6] in 1952, and is able to solve systems of linear equations with symmetric and positive definite coefficient matrices. First, it was merely considered a direct solver, and because it is more expensive in terms of computation steps than the Gaussian elimination, it was given little attention at the time. With the rise of powerful computers, the method became interesting once again as an iterative solver for computing an approximation of the solution [Fac00]. In the 1970s, it was modified in various ways, where the modifications often adapt to one specific problem, concerning the properties of the matrix A of the linear system $Ax = b$.

Among the large variety of Krylov subspace solvers, one important algorithm is the *GMRES Algorithm* (see Section 2.5.4), able to solve systems in which A is neither symmetric nor positive definite [Saa03, Bou01]. Since GMRES is able to solve any linear non-singular system, and can thus be applied to many different problems, it became the method of choice for many CFD applications [Bou01].

Aside from the individual characteristics of the different Krylov subspace methods, they are all based on algorithms generating orthogonal subspaces. One commonly used method to generate a Krylov subspaces is the Arnoldi Algorithm[7], with the Lanczos Algorithm[8] as a special case for symmetric matrices. While Axel Facius provides in [Fac00] a descriptive overview about the available Krylov subspace methods and the within used subspace generators, we summarize in the following sections only some aspects of the Arnoldi and Lanczos schemes and the CG and GMRES Krylov subspace solvers.

[4] Alexei Nikolaevich Krylov (⋆1863, †1945)
[5] Magnus Rudolph Hestenes (⋆1906, †1991)
[6] Eduard Stiefel (⋆1909, †1978)
[7] Walter Edwin Arnoldi (⋆1917, †1995)
[8] Cornelius Lanczos (⋆1893, †1974)

2.5.1. Arnoldi-Process

Arnoldi's method [Saa03] is an orthogonal projection method, that was originally introduced to reduce a dense matrix into its Hessenberg[9] form [Saa03]. Later it was discovered, that this algorithm simultaneously generates a sequence of Krylov subspaces, leading to an efficient technique for approximating eigenvalues of large sparse matrices.
Using the normalized vector v_j, the Krylov subspace is constructed successively. First, v_j is multiplied with the matrix A, then orthogonalized to all previous v_i by a conventional Gram[10]-Schmidt[11] Conjugation, and then normalized to get v_{j+1}. These iteratively constructed vectors $\{v_1, v_2 \dots v_m\}$ form the orthonormal m-th Krylov subspace. At the same time, the upper $(m+1) \times m$ Hessenberg matrix $\hat{H}_m$ is constructed out of the norm of the vectors v_i. The complete Arnoldi-Algorithm can be found in Algorithm 4.

Algorithm 4 Arnoldi's Algorithm [Saa03].

1: choose a starting vector v_1 with $\| v_1 \|_2 = 1$
2: **for** $(j = 1; j \leq m; j++)$ **do**
3: **for** $(i = 1; i \leq j; i++)$ **do**
4: compute $h_{i,j} = \langle Av_j, v_i \rangle$
5: **end for**
6: compute $w_j = Av_j - \sum_{i=1}^{j} h_{i,j} v_i$
7: $h_{j+1,j} = \| w_j \|_2$
8: **if** $(h_{j+1,j} == 0)$ **then**
9: stop
10: **end if**
11: $v_{j+1} = \frac{w_j}{h_{j+1,j}}$
12: **end for**

Implementing Arnoldi's method, one usually replaces the sum in line 6 and obtains the modified Algorithm 5.

Algorithm 5 Arnoldi's Algorithm (modified) [Saa03].

1: choose a starting vector v_1 with $\| v_1 \|_2 = 1$
2: **for** $(j = 1; j \leq m; j++)$ **do**
3: compute $w_j = Av_j$
4: **for** $(i = 1; i \leq j; i++)$ **do**
5: compute $h_{i,j} = \langle w_j, v_i \rangle$
6: compute $w_j = w_j - h_{i,j} v_i$
7: **end for**
8: $h_{j+1,j} = \| w_j \|_2$
9: **if** $(h_{j+1,j} == 0)$ **then**
10: stop
11: **end if**
12: $v_{j+1} = \frac{w_j}{h_{j+1,j}}$
13: **end for**

[9] Karl Hessenberg (⋆1904, †1959)
[10] Jørgen Pedersen Gram (⋆1850, †1916)
[11] Erhard Schmidt (⋆1876, †1959)

From the mathematical point of view, the two algorithms are equivalent in the case of non-existing rounding errors. As rounding errors usually occur, the second algorithm is superior, as it is numerically more stable [Saa03]. However, it may happen that the rounding errors are that severe, that still in the modified algorithm the orthogonalization steps are not sufficient to obtain proper results. In this case, the Gram-Schmidt orthogonalizer can be replaced by a Householder[12] scheme or by a Givens[13] rotation [GVL96], which both owns even more numerical stability. Another very efficient workaround would be to apply an additional Gram-Schmidt orthogonalization step which, according to Kahan's "twice is enough" [Par80] algorithm, necessarily provides a subspace of orthogonal vectors [GLRE05]. The Krylov-subspace and the Hessenberg matrix generated by the Arnoldi-Process have some pleasant properties, that are useful for the convergence analysis of the algorithms based on the Arnoldi-Process. Since we only want to give a brief overview, we limit this section to the facts that can be obtained, and refer to Yousef Saad's book [Saa03] for further information and the convergence proofs.

(1) Assuming the Arnoldi algorithm does not stop before the m-th step, then the generated subspace $\{v_1, v_2 \dots v_m\}$ forms an orthonormal basis of the Krylov subspace $\mathcal{K}_m = \{v_1, Av_1, A^2v_1, \dots A^m v_1\}$.

(2) If we denote the $n \times m$ matrix with column vectors $v_1, v_2 \dots v_m$ with V_m, the $(m+1) \times m$ Hessenberg matrix whose nonzero entries $h_{i,j}$ are defined by the Arnoldi algorithm with $\hat{H}_m$, and denote the matrix obtained from $\hat{H}_m$ by deleting its last row with H_m, the following properties hold:

$$\begin{aligned} AV_m &= V_m H_m + w_m e_m^T = V_{m+1}\hat{H}_m, \\ V_m^T A V_m &= H_m. \end{aligned}$$

(3) Arnoldi's algorithm breaks down at step j (i.e. $h_{j+1,j} = 0$) if and only if the minimal polynomial of A is of degree j. Moreover, in this case the subspace $\mathcal{K}_j$ is invariant under A.

2.5.2. Lanczos Algorithm

In the special case of a symmetric matrix A, Arnoldi's algorithm can be simplified as one obtains from

$$H_m = V_m^T A V_m = (V_m^T A^T V_m)^T = H_m^T$$

that $\hat{H}_m$ is symmetric too [Saa03]. As then the matrix $\hat{H}_m$ is a symmetric upper Hessenberg matrix, it even is a symmetric tridiagonal matrix. With $\alpha_j = h_{jj}$ and $\beta_j = h_{j-1,j}$ the matrix $\hat{H}_m$ reduces to

$$\hat{H}_m := \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix}. \tag{2.35}$$

Taking advantages of the special matrix layout of $\hat{H}_m$, we obtain the *Lanczos Algorithm* (see Algorithm 6, [Saa03]) where the orthogonalization of the new basis vector against all the previous basis vectors can be reduced to the orthogonalization against the last two vectors. This leads to large performance advantages, both in terms of necessary computation steps and memory need.

[12] Alston Scott Householder ($\star$1904, †1993)
[13] James Wallace Givens ($\star$1910, †1993)

Algorithm 6 Lanczos Algorithm [Saa03].

choose a starting vector v_1 with $||v_1||_2$=1, $\beta_1 = 0$,$v_0 = 0$
for $(j = 1; j \leq m; j++)$ **do**
  compute $w_j = Av_j - \beta_j v_{j-1}$
  compute $\alpha_j = \langle w_j, v_j \rangle$
  compute $w_j = w_j - \alpha_j v_j$
  compute $\beta_{j+1} = ||w_j||_2$
  if $(\beta_{j+1} == 0)$ **then**
    stop
  end if
  $v_{j+1} = \frac{1}{\beta_{j+1}} w_j$
end for

2.5.3. Conjugate Gradient Method

The Conjugate Gradient method can be considered as an iterative algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite [Saa03]. Such systems arise regularly when solving partial differential equations numerically [Bra07].
The algorithm can be motivated by using the Lanczos Algorithm to generate a Krylov subspace, and to expand Lanczos' method to obtain a solver for linear systems.

Algorithm 7 Conjugate Gradient Method [Saa03].

$x^0 := 0$
$r^0 := b - Ax^0$
$s^0 := r^0$
for $(k = 0; \| r^k \|_2 < \varepsilon \cdot \| r^0 \|_2; k++)$ **do**
  $\alpha_k = \frac{(r^k)^T r^k}{(s^k)^T A s^k}$
  $x^{k+1} = x^k + \alpha_k s^k$
  $r^{k+1} = r^k - \alpha_k A s^k$
  $\beta_{k+1} = \frac{(r^{k+1})^T r^{k+1}}{(r^k)^T r^k}$
  $s^{k+1} = r^{k+1} + \beta_{k+1} s^k$
end for

A detailed derivation of the Conjugate Gradient method (Algorithm 7) can be found in [She94], including a broad convergence analysis. We limit this section to one central result for the convergence rate: For a system of linear equations $Ax = b$ with a matrix A having uniformly distributed eigenvalues and the condition number $\kappa := \kappa_2(A)$, after

$$k = \left\lceil \frac{\sqrt{\kappa}}{2} \ln\left(\frac{2}{\varepsilon\kappa}\right) \right\rceil \tag{2.36}$$

iteration steps of the CG algorithm, the residual fulfills $\| r^k \|_2 \leq \varepsilon \| r^0 \|_2$. While this upper bound of course only holds for the case of non-existing rounding errors, in most cases the eigenvalues are clustered together and the algorithm converges even faster.

Concerning the computational effort of the CG algorithm, every iteration step is dominated by one matrix-vector multiplication. Additionally, it contains two scalar products and three vector updates, consisting of a vector addition and a scalar multiplication.
Every scalar product has the computational cost $2n-1$ and every vector update, consisting of a vector addition and a vector multiplication with a scalar, has the computational cost of $2n$. In the case of dense matrices, a matrix-vector multiplication has the computational cost $2n^2-n$ while for sparse matrices an upper bound is given by $2 \cdot nnz$, where nnz denotes the number of non-zeros. As this upper bound is also a reasonable approximation for the dense case, we can use it independently of the sparsity and obtain that the computational cost of performing one CG iteration loop can be approximated by

$$\underbrace{2 \cdot nnz}_{\text{matrix-vector-multiplication}} + \underbrace{2 \cdot (2 \cdot n - 1)}_{\text{two scalar products}} + \underbrace{3 \cdot (2 \cdot n)}_{\text{three vector updates (addition+multiplication)}} = 2 \cdot nnz + 10 \cdot n - 2. \tag{2.37}$$

Omitting the constant part, we can estimate the computational cost for the complete CG solver performing (2.36) iteration loops to guarantee a residual error smaller than $\varepsilon \parallel r_0 \parallel_2$ with [She94]

$$C_{\mathrm{CG}}(\varepsilon) = \left\lceil \frac{\sqrt{\kappa}}{2} \ln\left(\frac{2}{\varepsilon\kappa}\right) \right\rceil (2 \cdot nnz + 10n)\,. \tag{2.38}$$

2.5.4. GMRES Algorithm

The GMRES Algorithm (Generalized Minimum Residual Method) is another projection method working on Krylov subspaces which was developed by Yousef Saad and Martin Schultz for linear problems where the coefficient matrix A is neither necessarily symmetric nor necessarily positive definite [Saa03].

As GMRES uses Arnoldi's method to generate the Krylov subspace, and the entire Krylov subspace $\mathcal{K}_n$ spans $\mathbb{R}^n$, we note that for non-existing rounding errors an no early breakdown, GMRES evaluates the exact result after n steps [Saa03]. For large linear systems, problems may occur performing the whole algorithm due to a linear increase in computational and storage cost, and due to the loss of orthogonality of the Krylov subspaces triggered by rounding errors. Because choosing a small number $m < n$ of iterations often already provides a good approximation of the result, one normally uses GMRES as an iterative solver, with a stopping criterion depending on the residual norm. A detailed derivation and description of the GMRES method shown in Algorithm 8 can be found in the work of Yousef Saad [Saa03].

One drawback of Algorithm 8 is that the solver does not provide an approximation of the solution at each iteration step. Thus, it is possible to set an explicit number of steps that should be performed, but it remains difficult to estimate the accuracy of the solution a priori. Additionally, the algorithm does not entail the solution of the minimization problem $\parallel \beta e^1 - \hat{H}_m y \parallel_2$ so far. Saad describes a workaround that eliminates both drawbacks at the same time by applying rotations to the Hessenberg matrix $\hat{H}_m$, resulting in an upper triangular system [Saa03]. Using a Givens rotation for orthogonalization, the obtained algorithm including a residual stopping criterion and a solution computation is given in Algorithm 9.

One problem still remains in the modified variant too: the algorithm needs to store the whole Krylov basis until the residual has reached the preset residual threshold. For large linear systems, this may become very expensive in terms of memory and computational effort concerning the solving process of $H_m y = d$. To avoid this, one often uses a variant

Algorithm 8 GMRES Algorithm [Saa03].

1: compute $r^0 = b - Ax^0, \quad \beta = \| r^0 \|_2, \quad v_1 = \frac{r^0}{\beta}$
2: Define the $(m+1) \times m$ matrix $\hat{H}_m = \{h_{i,j}\}_{1 \leq i \leq m+1, 1 \leq j \leq m}$. Set $\hat{H}_m = 0$.
3: **for** $(j = 1; j \leq m; j++)$ **do**
4: compute $w_j = Av_j$
5: **for** $(i = 1; i \leq j; i++)$ **do**
6: $h_{i,j} = \langle w_j, v_i \rangle$
7: $w_j = w_j - h_{i,j} v_i$
8: **end for**
9: $h_{j+1,j} = \| w_j \|_2$
10: **if** $(h_{j+1,j} == 0)$ **then**
11: set $m = j$ and go to line 15
12: **end if**
13: $v_{j+1} = \frac{w_j}{h_{j+1,j}}$
14: **end for**
15: compute y_m, the minimizer of $\| \beta e^1 - \hat{H}_m y \|_2$
16: compute the approximation $x^m = x^0 + V_m y_m$

called *RESTART-GMRES*, or *GMRES-(m)* [Saa03]. The difference to the plain GMRES algorithm is that in case of the restart-variant, the computation of the Krylov subspace and the approximation is not executed until the residual has reached the threshold, but restarted after a certain number of steps m. The restart decreases the memory demand and the computational effort since the linear problem $H_m y = d$ stays at a lower dimension and only m Krylov subspace vectors have to be stored. A second advantage is that the orthogonality of the computed Krylov subspace is preserved to a higher grade, since the rounding effects are downscaled due to the restart of the Krylov-subspace generator. Before every restart of the algorithm, the solution approximation is updated and the stopping criterion is checked. Vincent Heuveline and Miloud Sadkane have proposed in [HS96] an efficient technique to enhance the method by polynomial acceleration: The underlying idea is to improve eigenvectors by choosing a polynomial that amplifies the components of the required eigendirections while damping those in the unwanted ones (also see [HS97b, HS97a]).

Like for the Conjugate Gradient, also for GMRES there exists some convergence analysis, see e.g. [Saa03, Bou01, SK06]. We limit this section to a special situation, where the matrix A of the linear system (2.1) is positive definite, all its eigenvalues lie somewhere in the ellipse $E(c, d, a)$, with center c, focal distance d, and major semi axis a, and that this ellipse excludes the origin. Furthermore, we assume A to be diagonalizable such that there exists a transformation X with $A = X \Lambda X^{-1}$ for $\Lambda = diag\{\lambda_1, \lambda_2, \ldots \lambda_n\}$. In this case we can guarantee a residual error term smaller than $\varepsilon \| r_0 \|_2$ after

$$k = \left\lceil \frac{\ln\left(\frac{\kappa_2(X)}{\varepsilon}\right)}{\ln\left(\frac{c+\sqrt{c^2-d^2}}{a+\sqrt{a^2-d^2}}\right)} \right\rceil \tag{2.39}$$

iteration steps [Saa03].

Concerning the computational cost, like in the CG case, every iteration step of GMRES is dominated by the matrix-vector multiplication needed to generate the Krylov subspace. Additionally, the kth iteration loop includes $k+1$ scalar products and $k+1$ vector updates consisting of a vector addition and a scalar multiplication. Denoting the number of

Algorithm 9 GMRES Algorithm based on Givens rotation (Hanke-Bourgeois [Bou01]).

compute $r^0 = b - Ax^0$, $d^0 = \beta_0 = \| r^0 \|_2$, $v_1 = \frac{r^0}{\beta_0}$
for $(m = 1;; m++)$ **do**
  {iteration process of GMRES}
  compute $w_m = Av_m$
  for $(i = 1; i \leq j; i++)$ **do**
    {Arnoldi's method}
    $h_{i,m} = \langle w_m, v_i \rangle$
    $w_m = w_m - h_{i,m} v_i$
  end for
  $\omega = \| w_m \|_2$
  for $(i = 1, i < j, i++)$ **do**
    {apply former rotation to h_k}
    $\tilde{h} = c_i h_{i,m} + s_i h_{i+1,m}$
    $h_{i+1,m} = -s_i h_{i,m} + c_i h_{i+1,m}$
    $h_{i,m} = \tilde{h}$
  end for
  if $(\omega \leq |h_{m,m}|)$ **then**
    {compute new rotation}
    $t_m = \frac{\omega}{|h_{m,m}|}$
    $c_m = \frac{h_{m,m}}{|h_{m,m}|\sqrt{1+t_m^2}}$
    $s_m = \frac{t_m}{\sqrt{1+t_m^2}}$
  else
    $t_m = \frac{h_{m,m}}{\omega}$
    $c_m = \frac{t_m}{\sqrt{1+t_m^2}}$
    $s_m = \frac{1}{\sqrt{1+t_m^2}}$
  end if
  $h_{m,m} = c_m h_{m,m} + s_m \omega$ {apply the rotation to the rest of $\hat{H}_m$}
  $d^j = -s_m d^{m-1}$ {apply the rotation to the right-hand-side}
  $d^{m-1} = c_m d^{m-1}$
  if $(|d^m| \leq \varepsilon)$ **then**
    stop
  end if
end for
solve $H_m y = d$ with the Gauss-Algorithm
define the matrix $V_m = [v_1 \dots v_m]$
compute the approximation $x^m = x^0 + V_m y$

non-zeros within the matrix A with nnz, we get for the kth iteration step of GMRES a computational complexity of

$$\underbrace{2 \cdot nnz}_{\text{matrix multiplication}} + \underbrace{(k+1)(2 \cdot n - 1)}_{\text{k+1 scalar products}} + \underbrace{(k+1)2 \cdot n.}_{\text{k+1 vector updates (addition+multiplication)}} \tag{2.40}$$

Summing up the first k steps, and assuming large dimension n, we can approximate the complexity with [Saa03]

$$\sum_{i=1}^{k} (2 \cdot nnz) + (i+1)(4 \cdot n - 1) \approx 2k \cdot nnz + (k^2 + 3k + 2) \cdot 2 \cdot n. \tag{2.41}$$

Comparing with (2.37) we can conclude that GMRES is more expensive in terms of computational efforts than the CG Algorithm for the symmetric positive definite case, but at the same time able to solve non-symmetric systems [Bou01].
Using equation (2.39), we can approximate the total cost of the plain GMRES solver, that guarantees a residual error smaller than ε by

$$\begin{aligned} C_{\text{GMRES}}(\varepsilon) = 2 \left\lceil \frac{\ln\left(\frac{\kappa_2(X)}{\varepsilon}\right)}{\ln\left(\frac{c+\sqrt{c^2-d^2}}{a+\sqrt{a^2-d^2}}\right)} \right\rceil \cdot nnz \\ + \left(\left\lceil \frac{\ln\left(\frac{\kappa_2(X)}{\varepsilon}\right)}{\ln\left(\frac{c+\sqrt{c^2-d^2}}{a+\sqrt{a^2-d^2}}\right)} \right\rceil^2 + 3 \left\lceil \frac{\ln\left(\frac{\kappa_2(X)}{\varepsilon}\right)}{\ln\left(\frac{c+\sqrt{c^2-d^2}}{a+\sqrt{a^2-d^2}}\right)} \right\rceil + 2 \right) 2n. \end{aligned} \tag{2.42}$$

As already indicated, GMRES is due to the high computational and memory cost often replaced by the restart variant GMRES-(m). The problem is that due to the always recomputed Krylov subspaces in the GMRES-(m) algorithm, no convergence analysis is possible for the general case. Some estimations for special matrix properties can be found in [Saa03] and [Bou01], but as we do not want to limit our analysis to these relatively strict demands, we refrain from showing them in this place.

2.6. Multigrid Methods

Multigrid methods may be formulated as defect correction methods that attempt to find a solution approximation by using a sequence of problems that are similar with respect to their structure, but differ in having successively decreasing dimension (see e.g. [Tro00]and references therein). They are usually applied to problems occurring in the field of finite element or finite difference discretizations of elliptic partial differential equations. In this case, choosing different mesh sizes with decreasing granularity for the different discretizations of the continuous problem can be used to obtain a sequence of discretized problems with decreasing dimension. This enables splitting the approximation error into high and low frequency terms that can then be treated with different efficiency on the distinct grid levels.

Beside the geometric multigrid methods that rely on information about the grid that was used for the discretization process of the partial differential equation, there also exist algebraic multigrid methods that only take a linear system of equations as input data [McC87, Stu01]. They are interesting in cases where the discretization process is unknown, the grid is irregular, or the matrix was obtained from a different application than the discretization of a partial differential equation. Since algebraic multigrid methods are generally not based on grids in the geometrical sense, one also often denotes them as *multilevel methods*.

2.6.1. High- and Low-Frequency Error Smoothing

One motivation for multigrid methods can be taken from decomposing the error of a solution approximation into high- and low-frequency parts which can be handled by relaxation methods with different efficiency on the distinct grid levels. Especially, the higher-frequency components are eliminated considerably faster than the low-frequent ones [Tro00]. To show this, we consider the linear system of equations

$$A_h x_h = b_h \tag{2.43}$$

on the grid Ω_h, and approximate the solution using the Richardson[14] algorithm [Mei05]:

$$x_h^{k+1} = x_h^k + \theta\left(b_h - Ax_h^k\right) = \left(I_h - \theta_h A_h\right) x_h^k + \theta_h b_h \tag{2.44}$$

where $\theta \in (0,1]$ denotes the damping coefficient. For a symmetric, positive definite matrix A_h we have a system of ortho-normal eigenvectors $\left\{w_h^{(i)}, i = 1 \dots n_h\right\}$ (n_h denoting the dimension of grid Ω_h) with corresponding eigenvalues $\lambda_{\min}(A_h) = \lambda_1 \leq \dots \leq \lambda_{\max}(A_h) =: \Lambda_h$. Therefore, we can write the initial error in the form

$$e_h^0 := x_h^0 - x_h^* = \sum_{i=1}^{n_h} \varepsilon_i w_h^{(i)}, \tag{2.45}$$

such that for the error e_h^k after k iterations, the equation

$$e_h^k = \left(I_h - \theta_h A_h\right)^k e_h^0 = \sum_{i=1}^{n_h} \varepsilon_i \left(I_h - \theta_h A_h\right)^k w_h^{(i)} = \sum_{i=1}^{n_h} \varepsilon_i \left(1 - \theta_h \lambda_i\right)^k w_h^{(i)}$$

holds. Hence,

$$\| e_h^k \|_2^2 = \sum_{i=1}^{n_h} \varepsilon_i^2 \left(1 - \theta_h \lambda_i\right)^{2k}. \tag{2.46}$$

Note that the condition $0 < \theta_h \leq \Lambda_h^{-1}$ is sufficient to guarantee the convergence of the Richardson iteration [Tro00]. Since then $|1 - \theta_h \lambda_i| << 1$ for large λ_i, and $|1 - \theta_h \lambda_1| \approx 1$, the high-frequency components of the error are smoothed faster than the low-frequency ones [Ran08]. Since the same applies to the residuals $r_h^k = b_h - A_h x_h^k = A_h e_h^k$, a low number of iterations already provides a residual satisfying

$$\| r_h^k \|_2^2 = \sum_{i=1}^{[n_h/2]} \varepsilon_i^2 \lambda_i^2 \left(1 - \theta_h \lambda_i\right)^{2k}, \tag{2.47}$$

where $[n_h/2] := \max\{n_h \in \mathbb{N} | n_h \leq \frac{n_h}{2}\}$. Furthermore, as the iterated defect term r_h^k is smooth on the grid Ω_h, a good approximation can be found on the grid Ω_{2h} with grid size $2h$. While Hackbusch shows in [Hac85] similar results for the Gauss-Seidel method, one also often replaces the Richardson algorithm by (weighted) Jacobi or SOR due to their superior smoothing properties [Tro00]. In Section 4.11 we will later investigate whether also the Block-asynchronous iteration we derive in Section 4.2 is a suitable replacement, and for this purpose compare to implementations using Gauss-Seidel smoothers.

[14] Lewis Fry Richardson (⋆1881, †1953)

2.6.2. Multigrid Methods in the Finite Element Context

To derive a multigrid method, we assume a sequence of grids Ω_{h_l}, $l = 0, \ldots, L$ with discretization fineness $h_0 > h_1 > \cdots > h_L$ and corresponding finite element spaces $V_l := V_{h_l} \subset V$ where $V = H^1(\Omega)$ denotes the continuous Banach space in which the partial differential equation we target is given. For convenience, we furthermore assume these finite element spaces to form a hierarchical sequence $V_0 \subset V_1 \subset V_2 \subset \cdots \subset V_L$. Although this assumption is not a necessary condition for multigrid methods, it simplifies the further analysis. For the case of a one-dimensional problem and a grid sequence where every second node is omitted on the successively coarser level, the grid sequence is visualized in Figure 2.3a.

If we now write the partial differential equation in the weak form

$$a(u, \varphi) = (f, \varphi) \quad \forall \varphi \in V, \tag{2.48}$$

we can obtain for the discretized form on the grid Ω_l

$$a(u_l, \varphi_l) = (f, \varphi_l) \quad \forall \varphi_l \in V_l.$$

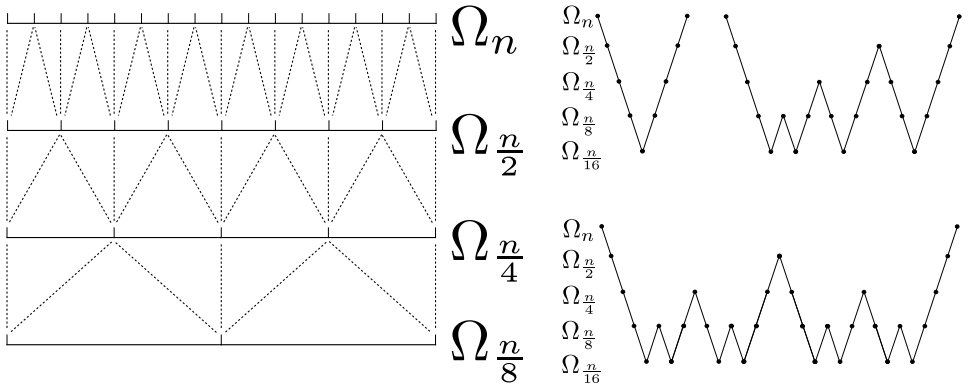
We now want to derive an iterative process, generating a sequence of successively better solution approximations on the finest grid Ω_L. In a first step, ν_1 iterations of the smoother are applied, which guarantees a smooth error on the coarser grid. While we analyzed the Richardson method with respect to its smoothing property in the last section, we mention again that also other relaxation schemes like Jacobi or Gauss-Seidel may be applied providing similar smoothing properties [Tro00]. We then compute the residual on the coarser grid Ω_{L-1} and obtain an error correction equation. The error correction term can then be computed using a direct method, an iterative solver, or be approximated by again applying an error correction scheme and successively coarser grids $\Omega_{L-2}, \ldots \Omega_0$. Afterwards, the error correction term is used for the solution update. The obtained solution approximation may be improved furthermore, e.g. again by applying another ν_2 relaxation steps of the smoother. Finally, the solution approximation is taken for the next iteration step, which completes one cycle of the multigrid method on the grid level L.
Every of these cycles therefore consists of $\nu_1 + \nu_2$ iteration steps of the smoother, the solution of the defect correction problem on the coarser grid and the grid operations. Due to the coarser grid, and the connected smaller dimension of the linear system, the resulting error correction equation is computationally less expensive to solve. This process can iteratively be cascaded to the coarsest grid, where the error correction equation has finally to be solved by applying either an iterative or a direct solver. The most important parts of a multigrid method are therefore the smoothing operations, the transfer operations between the different grid levels and the solver on the coarsest grid.

Aiming for the derivation of a multigrid algorithm in general form we use on the grid Ω_l the operator $\mathcal{A}_l : V_l \to V_l$ associated to the matrix $A_l = A_{h_l}$ by

$$(\mathcal{A}_l v_l, w_l) = A(v_l, w_l) = \langle A_l y_l, z_l \rangle \quad \forall v_l, w_l \in V_l. \tag{2.49}$$

Furthermore, we denote the smoothing operator with $\mathcal{S}_l(\cdot)$ where in case of using the Richardson iteration $\mathcal{S}_l = \mathcal{I}_l - \theta_l \mathcal{A}_l$. Finally, we introduce the transformation operations between the grid levels:

$$r_l^{l-1} : V_l \to V_{l-1} \quad \text{(Restriction)}$$
$$p_{l-1}^{l} : V_{l-1} \to V_l \quad \text{(Prolongation)}$$



(a) Sequence of successively coarser grids.

(b) Visualizing V-cycle (upper left), W-cycle (lower) and F-cycle (upper right) [Tro00].

Figure 2.3.: Basic multigrid principles.

In the finite element context with cascaded grid levels, the restriction r_l^{l-1} is the L^2-projection from Ω_l into Ω_{l-1}, and the prolongation p_{l-1}^l is the identity embedding [Ran08]. In the special case of solving the error correction equation exact on the grid Ω_{L-1}, the method is called *two-grid iteration method* [Tro00]. Usually, the process is iteratively applied to successively coarser grids, motivating for the term *mutigrid method*. Still, there exist different schemes how to organize the process as the structure is determined by the number of error correction computations on every grid level (we denote it by R). Implementations usually choose $R = 1$ or $R = 2$ [Wes92]. The resulting multigrid iteration schemes are called *V-Cylce* and *W-Cycle*, respectively. While the V-Cycle is usually very efficient in terms of computational cost, it shows inferior convergence properties compared to the computationally more expensive W-cycle [Tro00]. For $R \geq 3$, the multigrid iteration method becomes inefficient. A trade off between V-Cycle and W-Cycle is the *F-Cycle* shown in Figure 2.3b, which achieves convergence rates almost similar to the W-Cycle while reducing the computational effort. For the introduced notation, the multigrid method for solving the equation

$$\mathcal{A}_L u_L = f_L \tag{2.50}$$

on the finest grid $\mathcal{T}_L$ is is given in Algorithm 10.

Algorithm 10 Basic multigrid method recursively solves $A_l x_l = b_l$ at each level l, using restriction $r_l^{l-1}(\cdot)$, prolongation $p_{l-1}^l(\cdot)$ and smoother $\mathcal{S}_l^\nu(\cdot)$ operations [Tro00].

```
MultiGrid(x_l, b_l, l)
if (l == 0) then
    Solve A_l x_l = b_l            {exact solution on coarsest grid}
else
    x_l = S_l^{ν1}(x_l, b_l)       {pre-smoothing}
    r_{l-1} = r_l^{l-1}(b_l − A_l x_l)   {restriction}
    v_{l-1} = 0
    for (j = 0; j < R; j++) do
        MultiGrid(v_{l-1}, r_{l-1}, l − 1)   {coarse grid correction}
    end for
    x_l = x_l + p_{l-1}^l(v_{l-1})   {prolongation of coarse grid correction}
    x_l = S_l^{ν2}(x_l, b_l)       {post smoothing}
end if
```

In [Ran08] Rolf Rannacher provides a descriptive convergence analysis for the multigrid method in the finite element context and the Richardson operator as smoother. He first defines a smoothing and approximation property of the two-grid method, and then extends the results to multiple levels by induction. For multigrid methods using Jacobi or Gauss-Seidel smoothers, convergence results can for example be found in [Hac85] (also see [Tro00, Wes92] for more detailed background). Furthermore, there exist some complexity estimations showing that the computational cost for solving a problem with n unknowns using multigrid stays in the order of $\mathcal{O}(n)$ [Ran08, Tro00]. Hence, multigrid methods are among the most efficient solvers for systems of linear equations arising from finite element discretizations [Wes92, WLB00].

3. Asynchronous Iteration

3.1. Asynchronous Iteration

With the advent of computing systems consisting of multiple processors, the formerly used numerical methods and algorithms have to be adapted to the challenge of efficient parallel execution. An important concept in this framework is *Load Balancing* which aims for the equal distribution of the work among the processors. This technique is expected to increase the performance by minimizing the waiting time of components that have already finished their tasks [Rab89]. One colorful example is the geometric domain decomposition applied to a physical problem, where the subdomains are handled by the different processors [QV99].

A totally different approach is the idea of *Asynchronous Methods*, where idle times are avoided by eliminating synchronization points [FS00]. Since these methods imply neither the exact scheduling of the tasks nor the reproducibility of an application run, asynchronous methods are sometimes also denoted as *Chaotic Methods* in literature. While it may happen that some processors perform extra computations that do not contribute to the algorithm convergence, these methods are very beneficial when the load is not well balanced, or when communication between the processors is expensive in terms of computing time or energy consumption [FS00].

In the class of asynchronous methods, the very general term *Asynchronous Iteration* describes algorithms, where a preset order of the computations forming an iteration method is not adhered to. A comprehensive overview about the research conducted in this field is presented by Andreas Frommer and Daniel Szyld in [FS00], including linear methods based on matrix splitting, nonlinear update functions as well as methods applied to nonlinear systems of equations. While in literature, the term asynchronous iteration refers to general class of iteration schemes, we focus in this thesis on component wise relaxation methods based on the Jacobi splitting that lack any specific update order. Since the publication of the pioneering paper in 1969 by Chazan and Miranker [CM69], the topic of asynchronous iteration has been subject to research by many authors. Different papers contributed to the establishment of a comprehensive convergence theory [FS00, Bau78, BE86] and runtime results revealed the performance superiority for specific problems [ÜD86], [ÜD96]. Besides, timing models were developed to approximate the algorithm performance for a certain hardware and problem configuration [BSS99, DB91]. Despite the fact that asynchronous iterations are not considered to belong to the mainstream numerical methods, especially with the rise of heterogeneous workstation clusters they increasingly also come

into focus of High-Performance-Computing [EBSMG96, Fro98, EBFS05]. The purpose of the next sections is to present some theoretical background necessary to understand the idea of the block-asynchronous iteration for GPUs we introduce in Chapter 4 and its convergence properties. As a convention, we from now on use the term asynchronous iteration exclusively for asynchronous relaxation schemes based on the linear update function obtained from the Jacobi splitting and applied to a linear problem. To derive this method, we will first recall the Jacobi algorithm we already introduced in Chapter 2, and then introduce the asynchronous iteration by the approach that was also taken by Chazan and Miranker in "Chaotic Relaxation" [CM69]. Additionally, we provide some convergence analysis that will serve as basis for the convergence theory we establish for the weighted block-asynchronous iteration in Section 4.10.2.

3.1.1. Jacobi Method

In Chapter 2 we introduced component wise relaxation methods for solving systems of linear equations of the form (2.1)

$$Ax = b.$$

We want to remind that the Jacobi method (2.18)

$$x^{k+1} = Bx^k + d$$

with $B = I - D^{-1}A$ and $d = D^{-1}b$ can also be written in the component wise form (2.19)

$$\begin{aligned} x_i^{k+1} &= \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j^k \right) \\ &= \sum_{j=1}^{n} b_{i,j} x_j^k + d_i \end{aligned}$$

where

$$B = (b_{i,j}) = \begin{pmatrix} 0 & -\frac{a_{1,2}}{a_{1,1}} & -\frac{a_{1,3}}{a_{1,1}} & \dots & -\frac{a_{1,n}}{a_{1,1}} \\ -\frac{a_{2,1}}{a_{2,2}} & 0 & \ddots & \vdots & \\ -\frac{a_{3,1}}{a_{3,3}} & \ddots & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \\ -\frac{a_{n,1}}{a_{n,n}} & & \dots & & 0 \end{pmatrix}$$

and $d_i = \frac{b_i}{a_{i,i}}$. This implies that for the classical Jacobi method, the component updates have to be synchronized: no component can be updated twice before all other components are updated.

If this update order of the iteration process is not adhered to, i.e., the individual components are updated independently and without consideration of the current state of the other components, we denote the resulting algorithm as an asynchronous iteration method.

3.1.2. Traditional Approach to Asynchronous Iteration

We follow the traditional approach to asynchronous iteration methods, taken by Chazan and Miranker in their pioneering paper on this topic in 1969 [CM69]. Therefore, we may consider the iterative solution process of the system of linear equations (2.1)

$$Ax = b,$$

by using the Jacobi iteration (2.19)

$$x^{k+1} = Bx^k + d$$

asynchronously. The relaxation process can be defined as a sequence of solution approximations $(x^t)_{t=0,1\ldots}$ in which each component is updated by a function of all components. Without loss of generality, we from now on assume $b = 0$ in (2.1), and hence also obtain for the additive component $d = 0$. Furthermore, we assume the iteration steps t to be integer timesteps, where at each time step t exactly one component is updated ($t = 1, 2\ldots$). Note, that the iteration index t now has a different meaning compared to the former used m, as every component update already increases the iteration counter. We now introduce an update function $u(\cdot)$ and a shift function $s(\cdot,\cdot)$. For each non-negative integer time step t, the component of the solution approximation x that is updated at step t is given by $u(t)$. In the update at step t, the m-th component x_m of the solution approximation used in this step is $s(t,m)$ steps back. For a reasonable implementation, and also for showing convergence, it is necessary to have an upper bound $\bar{s}$ for this shift function $s(t,m) \leq \bar{s},\ \forall m \in \{1\ldots n\},\ \forall(t = 1, 2\ldots)$.

While $x^{m+1} = Bx^m$ may provide general information about the iteration scheme without taking into account the individual component updates, it may be beneficial to break down the iteration process, and to define an iteration matrix for the update of the i-th component, exclusively. The motivation is, that in every step of asynchronous iteration only one component is updated. To obtain such an iteration matrix which updates only component i, we define Z_i to be the zero matrix with the diagonal element $z_{ii} = \frac{1}{a_{ii}}$. Let now the component updated in step $t+1$ be i, then the iteration for this update reads:

$$\begin{aligned} x^{t+1} &= (I - Z_i A)\, x^t \\ &= \begin{pmatrix} 1 & 0 & 0 & \ldots & \ldots & 0 \\ 0 & 1 & 0 & & & 0 \\ \vdots & & \ddots & & & 0 \\ -\frac{a_{i1}}{a_{ii}} & -\frac{a_{i2}}{a_{ii}} & \ldots & 0 & \ldots & -\frac{a_{in}}{a_{ii}} \\ \vdots & \vdots & & & & \vdots \\ 0 & 0 & \ldots & \ldots & 0 & 1 \end{pmatrix} x^t \end{aligned} \tag{3.1}$$

If we denote with x^ν the approximation after ν local iterations starting from x^0, we have for the iterate

$$x^\nu = (I - Z_{i_\nu} A)\ldots(I - Z_{i_3} A)\,(I - Z_{i_2} A)\,(I - Z_{i_1} A)\, x^0$$

where $i_1, i_2, \ldots i_\nu \in \{1, 2\ldots n\}$ and $i_1, i_2, \ldots i_\nu$ is the update order of the components.

Using the introduced notation for updating one component, we can define the asynchronous iteration method:

Definition 3.1.1. *Considering the iteration sequence in an asynchronous iteration, the value of the i-th component of the solution approximation at time $t+1$ is defined as*

$$x_i^{t+1} = \begin{cases} (I - Z_i A) \cdot \begin{pmatrix} x_1^{t-s(t+1,1)} \\ \vdots \\ x_n^{t-s(t+1,n)} \end{pmatrix}, & \text{if} \quad u(t+1) = i \\ x_i^t, & \text{if} \quad u(t+1) \neq i \end{cases} \tag{3.2}$$

where $x_i^0 = x_{i,0}\ \forall i = 1, 2\ldots n$ ($x_{i,0}$ is the initial guess) and $s(\cdot,\cdot)$ denotes the introduced shift function.

This representation also provides information about how old the respective components used in the update function are. Depending on the memory architecture, this may differ from the information about the last update if the updated value of component i is not immediately communicated, such that the update of another component uses the former value. A descriptive example for this situation is the parallel update of the component values by a multicore processor, where each processing unit owns a local memory, and the information of the updated values has to be transmitted first [FS00]. An implementation of the iteration scheme (3.2) is given in Algorithm 11.

Algorithm 11 Asynchronous Iteration [FS00].

for all ($i \in \{1 \dots n\}$) **do** {asynchronous update order}
  read x from global memory
  $x_i^{new} = (I - Z_i A)x$
  overwrite x_i in global memory with x_i^{new}
end for

3.2. Convergence of Asynchronous Iteration

First we will provide a short overview about the convergence theory for asynchronous methods in general. Then we will state the convergence theory for the specific case of a nonsingular linear system of equations and a iteration matrix derived from Jacobi splitting. A comprehensive treatment can also be found in [FS00].

The first convergence theorems were derived by Chazan and Miranker for nonsingular linear systems, where the iteration matrix is derived from the Jacobi splitting. We will state these theorems, including the respective proofs, in Section 3.2.1, since we will need them for showing convergence of the weighted variants we derive in Section 4.10.2. Later, these convergence results were generalized by different papers establishing a convergence theory using a more general approach to asynchronous iteration based on a sequence of contracting subsets, see e.g. [BE86, Ber89, ÜD86, Bau78]. While there also exists some theory on the convergence of asynchronous methods based on block Jacobi (see [FS00]), El Baz was in [EB90] able to show global convergence for the case of having an M-function (see [OR70]) as iteration operator.

Apart from considering linear systems of equations, some work was also published on asynchronous iteration applied to nonlinear systems. Among the most relevant results are beside the generalizations of Tarazi [ET82] the achievements for the quasi-Newton schemes (e.g. see [Boj84, Xu99]).

3.2.1. Convergence Theory for Asynchronous Jacobi applied to Nonsingular Systems of Linear Equations

Using the notation given in Definition 3.1.1 introduced in Section 3.1, we now analyze the convergence characteristics for an iterative scheme of the form (2.12)

$$x_i^{k+1} = \sum_{j=1}^{n} b_{i,j} x_j^k + d_i, \quad i = 1, 2 \dots n$$

where the iteration matrix B is obtained from the Jacobi splitting, and again without loss of generality we set $d_i = 0$ for all $i = 1, 2 \dots n$.

Furthermore, we assume that the following conditions are fulfilled [Str97]:

Condition 3.2.1. *(a) Every component is updated infinitely often, and therefore, the update function $u(\cdot)$ takes on each value l, $1 \leq l \leq n$ infinitely often.*

(b) The shift function $s(\cdot,\cdot)$ is bounded by some $\bar{s}$ such that $0 \leq s(t,m) \leq \bar{s}$ $\forall t$, $\forall m$. For the initial step to be well defined, we furthermore require $s(t_0,m) \leq \bar{s}$ as well.

Theorem 3.2.2. *[CM69] Suppose*

$$x_i^{k+1} = \sum_{j=1}^{n} b_{i,j} x_j^k + d_i, \quad i = 1, 2 \dots n$$

is an asynchronous iteration scheme where (a) and (b) of Condition 3.2.1 are fulfilled. If there furthermore exists a positive vector $v \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$, $\alpha < 1$ such that

$$|B|\, v \leq \alpha v,$$

then the sequence of solution approximations x^t is component wise convergent to x^*, the unique solution of (2.1). Especially, the distinct components x_i^t of the solution approximation converge to the respective solution x_i^*.

Proof. [CM69] We recall that we assumed without loss of generality that $b = 0$ in (2.1) and therefore $d_i = 0$ $\forall i = 1, 2 \dots n$. The theorem's proof is based on analyzing the difference between $x^* = 0$, the unique solution to (2.1) and the iterates x^t. Let $e^t = x^* - x^t$ be the error in the t-th iteration. We now consider the first $\bar{s}$ iterates in the process. We required that there exists $\alpha < 1$, $v > 0$ with $|B|\, v \leq \alpha v$. Since all components of v are positive, there exists a positive value γ such that $|e^t| \leq \gamma v$ for $0 \leq t \leq \bar{s}$ (component-wise, $|e_j^t| \leq \gamma v_j$ $\forall j = 1 \dots n$). We now consider any component i that is updated using any of these $\bar{s}$ vectors forming the first $\bar{s}$ iterates. Then, since $Bx_i^t = -Be_i^t$ ($\bar{x_i} = 0$ $\forall i = 1 \dots n$), the update satisfies

$$\begin{aligned} |e_i^{t+1}| = |x_i^* - x_i^{t+1}| &= |0 - x_i^{t+1}| \\ &= |\sum_{j=1}^{n} b_{i,j} x_j^{t-s(t,j)}| \leq \sum_{j=1}^{n} |b_{i,j} e_j^{t-s(t,j)}| \\ &\leq \sum_{j=1}^{n} |b_{i,j}||e_j^{t-s(t,j)}| \leq \alpha\gamma v_i. \end{aligned}$$

If t_1 is the first instance after $\bar{s}$ for which all components have been updated, then $|e^{t_1}| \leq \alpha\gamma v$. Moreover, $|e^t| \leq \alpha\gamma v$ for all $t \geq t_1$. Similarly, if t_2 is the next instance after t_1 for which all components have been updated a second time, then $|e^t| \leq \alpha^2\gamma v$ for all $t \geq t_2$. This way we obtain that the error $|e^t| = |x^* - x^t|$ converges to zero. Hence, the solution approximation is convergent to x^*, the unique solution.

□

Theorem 3.2.3. *[CM69] Suppose*

$$x_i^{k+1} = \sum_{j=1}^{n} b_{i,j} x_j^k + d_i, \quad i = 1, 2 \dots n$$

is an asynchronous iteration scheme where (a) and (b) of Condition 3.2.1 are fulfilled. If furthermore

$$\rho\left(|B|\right) < 1$$

where $\rho(|B|)$ is the spectral radius of the component wise non-negative matrix $|B|$, then there exists a positive $v \in \mathbb{R}^n$ and $\alpha < 1$ such that $|B|\, v \leq \alpha v$. Especially the sequence of solution approximations x^t is component wise convergent to x^*, the unique solution of (2.1).

Proof. [CM69] Due to Theorem 3.2.2 it is sufficient to show that if $\rho(|B|) < 1$, there exists a positive v and $\alpha < 1$ such that $|B|\, v \leq \alpha v$. While for irreducible matrices this statement is covered by the Perron-Frobenius Theorem [Fro12], we have to prove it for the case of B being reducible.

To achieve this, we first show that if a matrix F has the form

$$F = \begin{pmatrix} F_{1,1} & F_{1,2} \\ 0 & F_{2,2} \end{pmatrix} \tag{3.3}$$

where for some vectors $v_1, v_2 > 0$ and $\alpha_1, \alpha_2 \in \mathbb{R}$ the inequalities

$$\begin{aligned} F_{1,1} v_1 &\leq \alpha_1 v_1, \\ F_{2,2} v_2 &\leq \alpha_2 v_2 \end{aligned}$$

hold, then for any $\varepsilon > 0$ there exists v with $Fv \leq (\alpha + \varepsilon)v$, where $\alpha = \max\{\alpha_1, \alpha_2\}$.

Indeed, if we choose $v = (v_1, \gamma v_2)$, then

$$Fv \leq (\alpha_1 v_1 + \gamma F_{12} v_2, \gamma \alpha_2 v_2).$$

With γ sufficiently small we get $\alpha_1 v_1 + \gamma F_{12} v_2 < (\alpha + \varepsilon) v_1$, without impacting the approximation $\gamma \alpha_2 v_2 \leq \gamma \alpha v_2$.

Instead of showing that if $\rho(|B|) < 1$ there exists a positive v with $|B|v < \alpha v$, $\alpha < 1$, we show the property for the normal form of B in Lemma 3.2.4 by using the statement we just established. The reason therefore is, that the normal form $\tilde{B}$ of a matrix B can be derived by applying matrix multiplications of the form

$$PBP^T = \tilde{B},$$

where $P \geq 0$, $P^T P = I$, and

$$\begin{aligned} & Bv && \leq \alpha v \\ \Leftrightarrow \quad & PBv && \leq \alpha P v \\ \Leftrightarrow \quad & PBP^T Pv && \leq \alpha P v \\ \Leftrightarrow \quad & \tilde{B} Pv && \leq \alpha P v. \end{aligned}$$

Hence, we get that the permutation of rows and columns does not impact the desired matrix property. Especially, if the property holds for a matrix, it also holds for its normal form and vice versa, only the vector v has to be replaced by Pv.

Lemma 3.2.4. *Let $|B|$ be a component wise non-negative matrix with corresponding normal form $\tilde{B}$. Then, if $\rho(|B|) < 1$ there exists $v > 0$ with $|\tilde{B}|v < \alpha v$, $\alpha < 1$.*

Proof. We show this by induction on the number of elements in the normal form $\tilde{B}$ and the results from the Perron-Frobenius Theorem [Fro12, Hup90, Mey00]:

Induction base:
The proposition is true if $|\tilde{B}|$ is irreducible according to the Perron-Frobenius Theorem [Mey00, Var10].

Induction hypothesis:
Suppose it is shown for any $|\tilde{B}|$ whose normal form has q block components

$$|\tilde{B}| = \begin{pmatrix} \tilde{B}_{1,1} & \tilde{B}_{1,2} & \tilde{B}_{1,q} \\ & \ddots & \vdots \\ 0 & & \tilde{B}_{1,q} \end{pmatrix}.$$

Induction step:
[CM69] Suppose $|\tilde{B}|$ is not irreducible and has $n+1$ components. We can then write $|\tilde{B}|$ in the form

$$|\tilde{B}| = \begin{pmatrix} F_{1,1} & F_{1,2} \\ 0 & F_{2,2} \end{pmatrix}$$

where $F_{2,2}$ has n components and $F_{1,1}$ is irreducible. Then, $F_{1,1}v_1 \leq \alpha_1 v_1$. Furthermore, $F_{2,2}v_2 \leq \alpha_2 v_2$ by induction hypothesis, where $\rho(|\tilde{B}|) < 1$ and $\alpha_2 \leq 1$. Using the statement from above we get for some $\gamma > 0$, $v = (v_1, \gamma v_2)$:

$$|\tilde{B}| \leq \alpha v, \quad \alpha < 1$$

completing the proof.

□

As already indicated, if this property holds for the normal form of a matrix, we also have it for the original matrix B. Hence we can find a (possibly different) $v > 0$ with $|B|\, v < \alpha v$, $\alpha < 1$. □

Conversely, if $|B|\, v \leq \alpha v, v > 0, \alpha < 1$, B is a contraction in the norm defined by $\|x\| = \max\{\frac{|x_i|}{v_i}\}$. Hence, the spectral radius of $|B|$ is smaller than 1.

Chazan and Miranker also provide an extension to this theorem, stating that the convergence of the asynchronous iteration is remained when using suitable weights:

Theorem 3.2.5. *[CM69] Suppose the asynchronous iteration fulfilling (a) and (b) of Condition 3.2.1 is modified by using weights such that $B_\omega = (I - \omega D^{-1}A)$ and $d_\omega = \omega D^{-1}b$. If the spectral radius $\rho(|B|) = \alpha < 1$, then the weighted asynchronous iteration converges for all ω with $0 < \omega < \frac{2}{\alpha+1}$.*

Proof. We want to show that $\rho(|B_\omega|) < 1$ or alternatively that there exists $v > 0$ so that $|B_\omega|v < \beta v$, $\beta < 1$. For the case $\omega = 1$ we obtain from Theorem 3.2.3 that there exists $v > 0$ so that $|B_1|v < \alpha v$. But then

$$|B_\omega|v \leq (I|(1-\omega)| + \omega|B_1|)v \leq |(1-\omega)|v + \omega\alpha v = (|1-\omega| + \omega\alpha)v.$$

Let $\beta = (|1-\omega| + \omega\alpha)$. It remains to show that $\beta < 1$. If $1 < \omega < \frac{2}{\alpha+1}$, then $\beta = \alpha\omega + (\omega - 1) = (1+\alpha)\omega - 1 < 1$. On the other hand, if $0 \leq \omega \leq 1$ we have $\beta = \alpha\omega + (1-\omega) = (-(1-\alpha)\omega + 1) < 1$ since $\alpha < 1$.

□

Examples for systems of linear equations fulfilling the sufficient convergence conditions are [CM69, Var10]:

1. A symmetric and strictly diagonal dominant.
2. A irreducibly diagonal dominant.
3. A symmetric positive definite with non-positive off-diagonal entries.

Now that we have established some theory for the sufficient convergence condition, it may be interesting to focus on a necessary condition for convergence. But for this, general results are more involved. Chazan and Miranker provide in [CM69] an example where the asynchronous iteration does not converge if the condition $\rho(|B|) \leq 1$ is not fulfilled. This is achieved by using a special update pattern, where the shift function $s(t, m)$ depends on the component. Therefore, it can not be considered as a general result.

Strikwerda attempts to prove that $\rho(|B|) \leq 1$ is not only a sufficient but also necessary condition for the convergence of asynchronous iteration in [Str97]. The basic is to construct an artificial update pattern for which the method does not converge. Also Berteskas and Tsistiklis in [BT89] and Su et al. in [SBKK98] provide constructions for non-converging update patterns for the case $\rho(|B|) \geq 1$. One might now conclude, that $\rho(|B|) < 1$ is not only a sufficient but also a necessary condition for convergence.

Nevertheless, several papers, including Lubachevsky and Mitra [LM86], show the convergence of asynchronous iteration for systems, where the spectral radius of the non-negative matrix is exactly one (see also [Pot98]). This result for singular matrices representing Markov chains seems to contradict the proves based on non-converging sequences. In [Szy98b] David Szyld resolves this apperent contradiction. In the following, we want to summarize the most important aspects of this issue. First, let us assume B to be non-negative. Due to $A = D - (D - A)$ and $B = D^{-1}(D - A) = (I - D^{-1}A)$ we get that $A = D(I - B)$. We may conclude that A singular implies that 1 is an eigenvalue of B and $\rho(B) = 1$. Vice versa, if $\rho(B) = 1$, 1 is an eigenvalue of B and A is singular. Using this fact, we now analyze the proofs for the necessary condition.

They all share the assumption of the existence of a unique solution to the system of linear equations

$$(I - B)x = b,$$

i.e. they all assume $(I - B)$ to be nonsingular. But as we have just seen, this is not possible for $\rho(|B|) = 1$. In other words, the case $\rho(|B|) = 1$ is not covered by any of the constructions of non-converging sequences in literature. Only Su et al. treat the case $\rho(|B|) = 1$ separately [SBKK98]: The authors show that in this case, there exists a sequence of vectors not converging to the zero vector, but the limit is the Perron vector to which the convergence is desired [Szy98b]. Therefore, this case should not be included in the necessary condition statement.

To account for the fact that the proofs for the necessary condition all neglect the fact, that for $\rho(|B|) = 1$ the system $(I - B)$ becomes singular, Szyld reformulates the convergence condition for asynchronous iteration:

Theorem 3.2.6. *[Szy98b] If $\rho(|B|) < 1$, the asynchronous iteration fulfilling (a) and (b) of Condition 3.2.1 converges to the unique solution. If $\rho(|B|) = 1$, under certain conditions, convergence can be achieved. If $\rho(|B|) > 1$ and if 1 is not an eigenvalue of B, an initial vector x^0 and an update pattern can be constructed for which the asynchronous iteration does not converge.*

Hence, $\rho(|B|) < 1$ is only a sufficient but not a necessary condition for the convergence of asynchronous iteration.

3.3. Asynchronous Two-Stage Iteration

In order to adapt asynchronous iteration to modern hardware systems consisting of multiple processors it is reasonable to split the linear system into blocks, that can then be handled independently. This approach is similar to the block iterative solvers we introduced in Section 2.4.4. Using an iterative solver on the subsystems, the algorithm splits into a cascaded iterative method with an inner and an outer iteration. While we denoted the synchronous versions as two-stage iteration methods, allowing for asynchronism in either the inner or the outer or both iteration methods motivates the term *Asynchronous Two-Stage Iteration Methods* [BMPS99]. The synchronous block algorithms like block-Jacobi arise as special cases of this general class for a unique update pattern. While Bru, Elsner and Neumann propose in [BEN88] two different asynchronous two-staged methods, Frommer and Szyld generalize these ideas in [Fro94] and provide a more comprehensive overview about different classes of asynchronous two-staged methods. The main idea is to distinguish between cascaded iterations where either only the inner, or only the outer, or none of them are synchronized. Additionally, the communication of the updated components provides another parameter for classification.

In the following sections we want to summarize some theory available in the literature and extend it to a more comprehensive classification, especially we introduce the class of block-asynchronous iteration we analyze in Chapter 4.

3.3.1. Inner Asynchronous Two-Stage Method

We first consider the case, where the solution process of the sub-equations (2.30) is handled separately by different processors. While all iterations conducted to the sub-equations in the (k)-th outer iteration use the same outer values, equivalent to the beginning of the global iteration, they are independent in the sense that the iteration methods may be different. This includes the case where the inner iteration methods differ in the number of local iterations, like in the method "A" proposed by Bru, Elsner and Neumann [BEN88]. The local iteration count might either be prescribed in advance or it can be determined at each step using some inner convergence criteria like proposed by Elman and Golub in [GY97], Golub and Overton [GO11], [GO87] or Golub and Ye [EG93]. As an extension to the method "A" proposed in [BEN88], Frommer and Szyld allow in [BMPS99] to apply an inner iteration method that is derived by using a different matrix splitting than the outer iteration is based on, e.g. the situation where the outer block Jacobi utilizes Gauss-Seidel schemes (see Section 2.4.2) for solving the sub-equations.

The obtained algorithms are synchronous in the sense that the (k)-th global iteration cannot start until all block-components of the $(k-1)$-th global iteration have been updated. They may be seen as a special case of Algorithm 4.1 in [LRS90] of Lanzkron, Rose and Szyld, [BMPS99]. In [Fro94] Frommer and Szyld prove the convergence of this two-stage iteration schemes. Despite the fact that the iteration count for the individual components may no longer be consistent due to the different iteration methods on the subdomains, this class of two-stage iterations is usually not considered as asynchronous.

We now extend the model by allowing for asynchronous methods to solve the sub-problems. As this extension enables the asynchronous usage of multiple processors for the solution of the sub-equations, we denote the obtained class as *Inner Asynchronous Two-Stage Methods*. It is more general than the model "A" proposed by Bru, Elsner and Neumann [BEN88], as it implies a more independent choice of the iteration methods for the respective sub-problems, and more general as the block two-stage method in [Fro94], since it allows for asynchronous solvers on the sub-problems.

For these methods, the Algorithm 11 has to be modified: the former iteration function, we from now on denote it with $H_i(x) = I - Z_i A) \cdot x$, has to be adapted to the situation, where

the off-block components stay equal in the local iterations, but the iteration status of the local components may differ. Therefore, $H_i(x)$ is replaced by $K_i(x, y)$, where y denotes the local components in the subdomain of the iteration. Using this notation, we can obtain a model for the inner asynchronous two-stage method, see Algorithm 12.

Algorithm 12 Inner Asynchronous Two-Stage Iteration.

```
for (J = J_1; J < J_q; J++) do {synchronous outer loop}
  read x from global memory
  set y = x
  for (k = 0; k < iter_k; k++) do {different local stopping criterion}
    for all (i ∈ J_k) do {asynchronous local updates}
      y_i^new = K_i(x, y)
      y_i = y_i^new
    end for
  end for
  overwrite x in global memory with y
end for
```

3.3.2. Outer Asynchronous Two-Stage Method

We now consider a variation of the inner two-stage iteration, where not only the inner iteration may be asynchronous, but additionally the outer iteration is no longer synchronized [BMPS99]. A practical example is an implementation, where the processors solving the distinct sub-problems are allowed to write back their results to the main memory at any point in time, and always read in the most recent data for the next iteration. They are allowed to start the computation of the next iterate of the respective block-component without waiting for the simultaneous completion of the same iterate of the other block-components, and to write back their respective block at the end of the local iteration [BMPS99]. Hence, the previous iterate may no longer be available to all processors. To stress that the block-components of the solution approximation are updated using a vector consisting of block-components of different previous, not necessary the latest, iterates, we denote these algorithms with *Outer Asynchronous Two-Stage Methods*. They are similar to the model "B" in [BEN88].

An important property of the outer asynchronous two-stage methods, visualized in Algorithm 13, is the fact, that all component updates in one block use the same values for the off-block iterates, equal to the values at the beginning of the local iteration phase.

Algorithm 13 Outer Asynchronous Two-Stage Iteration.

```
for all (J_k ∈ {J_1 ... J_q}) do {asynchronous outer loop}
  read x from global memory
  set y = x
  for (k = 0; k < iter_k; k++) do {different local stopping criterion}
    for all (i ∈ J_k) do {asynchronous local updates}
      y_i^new = K_i(x, y)
      y_i = y_i^new
    end for
  end for
  overwrite x in global memory with y
end for
```

Note that the inner asynchronous two-stage method we analyzed in the last section is a special case, where the global updates are synchronized. Other examples for outer

asynchronous two-staged methods include the asynchronous block-Jacobi, proposed e.g. by Szyld in [Szy98a] and used as a preconditioner in [CG10]. All these asynchronous block-Jacobi methods share the property, that the inner iterations are synchronous. The block-asynchronous iteration we analyze in Chapter 4 arises as a special case of the asynchronous block-Jacobi, where also the inner iteration is based on a Jacobi splitting.

3.3.3. Block-Asynchronous Methods

One subclass of the outer asynchronous two-stage methods are the inner asynchronous two-stage methods, where the outer iteration is synchronized but the inner iteration may be asynchronous. Complementary to those we define the *Block-Asynchronous Methods* as another subclass, where the outer iteration may be asynchronous, but the inner iteration, based on the same matrix splitting, is synchronized (see Algorithm 14). The motivation for this approach is to use multiple processors to solve the subsystems in parallel with a preset update order, but to perform the global updates independent and without consideration of the iteration status of the other components. As this class perfectly fits the demands of implementations using graphic processing units, it becomes very interesting for modern High-Performance Computing, and we will put more focus on these methods in Chapter 4, where we base the outer and the inner iteration method on a Jacobi splitting.

Algorithm 14 Block-Asynchronous Iteration.

```
for all (J_k ∈ {J_1 ... J_q}) do {asynchronous outer loop}
  read x from global memory
  set y = x
  for (k = 0; k < iter; k++) do {equal local stopping criterion}
    for (i = J_k(begin); i < J_k(end); i++) do {synchronous local updates}
      y_i^new = K_i(x, y)
      y_i = y_i^new
    end for
    overwrite x in global memory with y
  end for
end for
```

3.3.4. Totally Asynchronous Two-Stage Methods

The only restriction of the outer asynchronous two-stage methods is, that the off-block iterates used for the updates in one block are all equal to the beginning of the local iterations on the sub-problem. This is equivalent to an implementation where the newly computed iterates are neither written back to the global memory until the iteration process on the sub-problem is completed, nor the off-block iterates change during the iteration process on the sub-problem. If we now drop these restrictions, we obtain algorithms we denote as *Totally Asynchronous Two-Stage Methods* [Fro94], see Algorithm 15. In these, the processors iterating a block take in each inner iteration the most recent iterates from the other block-components. This implies, that for the different component updates in a block, different values for the off-block entries of the solution approximation are used. All asynchronous two-stage methods we analyzed in the last sections are special cases of this class.

3.3.5. Overlapping Blocks in Asynchronous Two-Stage Methods

All two-stage methods are based on the idea of splitting the global problem into sub-problems that can then be handled independently. While this corresponds to a block decomposition of the system matrix, it is also analogous to a domain decomposition of the

Algorithm 15 Totally Asynchronous Two-Stage Iteration.

```
for all (J_k ∈ {J_1 ... J_q}) do {asynchronous outer loop}
  for (k = 0; k < iter_k; k++) do {different local stopping criterion}
    for all (i ∈ J_k) do {asynchronous local updates}
      read x from global memory
      set y = x
      y_i^new = K_i(x, y)        i ∈ J_j
      y_i = y_i^new
      overwrite x_i in global memory with y_i {during local iterations}
    end for
  end for
end for
```

discretized problem. An interesting question in this context is what happens, if some of the matrix blocks overlap. In domain decomposition methods, the concept of overlapping domains is often applied, since it allows for faster propagation of the information [HC03, Cai09]. Similar approaches may be beneficial when targeting the discretized problem by using overlapping blocks when solving the system of linear equations [CS96]. The models we introduced for the asynchronous two-stage iterations do in general not preclude the blocks to overlap. Only for the block-asynchronous iteration, a special case occurs: For overlapping blocks, the components that are part of several blocks are updated considerably more often than the other components in this block. Assuming the situation of k local updates on every sub-problem, components shared by two blocks are updated $2k$ times, while all other components are only updated k times in one global iteration. Like in domain decomposition methods, using overlapping blocks in the matrix splitting we expect the off-block entries in the matrix to have more influence, which may result in an improved convergence rate. In [FSS97] Frommer, Schwandt, and Szyld have shown that a certain degree of overlapping together with a scheme for combining different contributions within the overlap, potentially accelerates the overall iteration convergence (also see [BMR97]).

3.3.6. Convergence of Asynchronous Two-Stage Methods

In Table 3.1 we summarize the properties of the different classes of two-stage iterative methods visualized in Figure 3.1. We observe that having proven the convergence for the totally asynchronous two-stage methods implies the convergence of all other classes, since they arise as subsets. The convergence results for asynchronous iteration we stated in Section 3.2.1 do in general not apply for asynchronous two-stage methods: The different number of local component updates is crucial in the proofs for convergence [Fro94], as it is impossible to derive a unique iteration matrix.

However, in the unweighted block-asynchronous methods we have the same number of local component updates on the sub-problems. Hence, as long as the distinct blocks do not overlap, the convergence theory previously stated in Section 3.2.1 for asynchronous iteration also applies to the block-asynchronous iteration.

Since the main focus of this thesis is on these block-asynchronous iteration methods, we refrain from providing a comprehensive convergence theory for asynchronous two-stage iteration and refer to the results given by Frommer and Szyld in [Fro94].

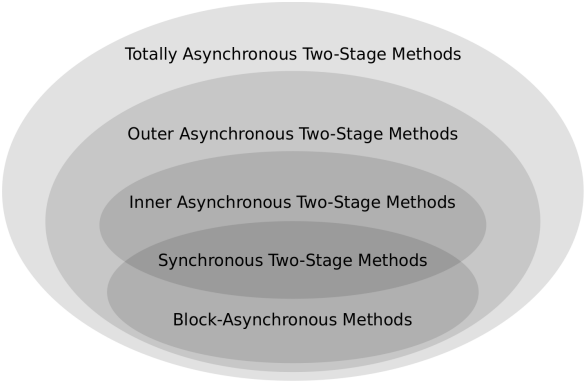


Figure 3.1.: Classification of two-stage iterative methods.

Method	inner iter.	outer iter.	values used for inner iter.
Sync. Two-Stage Iteration	sync.	sync.	beginning of iter.
Inner Async. Two-Stage Iteration	async.	sync.	beginning of iter.
Block-async Iteration	sync.	async.	beginning of iter.
Outer Async. Two-Stage Iteration	async.	async.	beginning of iter.
Totall Async. Two-Stage Iteration	async.	async.	latest available

Table 3.1.: Classification of two-stage iterative methods [BEN88, Fro94, ATDH12a].

4. Block-Asynchronous Iteration

Two-stage iteration methods are suitable for multiprocessors, where the different subproblems can be handled independently by distinct processing units (see Section 2.4.4). But at the same time, the necessary synchronizations between the iterations limit the level of parallelism when working on heterogeneous clusters, that seem to become more and more relevant in scientific High Performance Computing. To overcome these limitations, it may be reasonable to allow for asynchronous handling of the subproblems by different hardware devices. In this chapter we want to derive and analyze the properties of the *Block-Asynchronous Iteration Method* we already listed as one class of asynchronous two-stage iteration adapted to hardware systems accelerated by graphics processing units (GPUs). For this purpose, we start with the naive asynchronous iteration, then split, motivated by the hardware architecture, the linear system into blocks that can be handled independently. Using SIMD devices like GPUs, the component updates on these blocks can than be conducted with high efficiency. Adding local iterations on the subdomains we have come full circle to the asynchronous two-stage iterations.
As the non-deterministic behavior of asynchronous methods is an important aspect when using them in scientific computation, we dedicate Section 4.3 to quantify the impact of this issue. Using different parameter configurations, we then compare in Section 4.4 convergence and performance of the block-asynchronous iteration to synchronized methods. Afterwards, we also target multi-GPU systems in Sections 4.6/4.7. In Section 4.8 we derive an algorithm optimized for sparse systems and show performance results in Section 4.9. As the properties of the system of linear equations like sparsity pattern and diagonal dominance have significant impact on the convergence properties, we introduce in Section 4.10 different techniques that may be applied to adapt to a specific problem setup. This includes weighting techniques to account for diagnonal dominance (Section 4.10.2) as well as concepts for the method's adaption to the discretization of a partial differential equation (Section 4.10.6). Finally, we show in Section 4.11 how block-asynchronous iteration may be used to replace the traditionally applied smoothers in multigrid methods, and analyze in Section 4.12 the performance improvement when employing them as error correction solver in mixed precision iterative refinement methods [AHR10]. Further, we propose to handle also the residual computation asynchronously, and derive in Section 4.13 a block-asynchronous iterative refinement method. Finally we conclude this chapter with Section 4.14.2, where we use block-asynchronous iteration in the solution process of a nonlinear instationary partial differential equation arising in the simulation of pattern formation in mathematical biology.

However, in order to understand the fundamental idea of block-asynchronous iteration on GPUs we start in Section 4.1 with recalling some basic principles of GPU programming. While a more comprehensive overview about GPU programming can be found in [Göd10], the architecture of graphics processing units and of the heterogeneous clusters used in this thesis are given in Appendix A.

4.1. General Purpose GPU-Computing

While graphics processing units (GPUs) are primarily designed for one particular class of applications, the rasterization and depth-buffering based interactive computer graphics [Göd10], using them for general purpose computing (GPGPU) is increasingly becoming interesting, especially when targeting highly parallel applications [KBD10]. To understand why and how GPUs can be used in scientific computing, it is useful to provide some background about the historical evolution of graphics processing units, their architecture as well as programming paradigms. A more comprehensive discussion about the history of GPUs, the programming concepts, the different hardware architectures, and a list of the most relevant publications can be found in [Göd10].

As already stated, GPUs were originally designed for graphics purpose only. To meet the challenges that typically arise in graphics workloads, three properties can be identified, that make GPUs so different from general-purpose CPUs [Göd10]:

- Graphics workloads are inherently massively parallel. In fact, the huge performance demands can only be satisfied by exploiting this parallelism to extreme scales, at least compared with other (single-chip) processors. Basically, the GPU architecture is centered around a large number of fine-grained parallel processors.
- The bandwidth demand of graphics tasks, in particular multi-texturing with advanced, anisotropic filtering, are insatiable. Furthermore, the memory subsystem must be able to serve many concurrent requests.

These requirements ask for a very special architecture and programming concept, the so-called *graphics pipeline*. The hardware accounts for this concept by a high core number organized in several arrays, called *streaming multiprocessors*, that enable the simultaneous execution of an operation on a large set of data [OLG+07, Göd10, KBD10].

This concept also removes the necessity of expensive control logic to tackle typical hazards induced by instruction level parallelism such as read-after-write, write-after-read, synchronization, deadlocks and other race conditions [Göd10]. To maximize throughput over latency, the pipeline is very deep, with thousands of primitives in flight at a time. In a CPU, any given operation may take on the order of 20 cycles between entering and leaving the processing pipeline (assuming a level-1 cache hit for data); on the GPU, in contrast, operations may take thousands of cycles to finish. In summary, the implementation of the graphics pipeline in hardware allows to dedicate a much larger percentage of the available transistors to actual computation rather than to control logic, at least compared with commodity CPU designs [Göd10].

In the programming model, the graphics pipeline is reflected as data parallelism which allows to conduct the same operation, usually denoted as kernel operation, on a large set of data in parallel. Since it is impossible to execute different operations by the distinct processing units (cores) at the same time, GPUs may be considered to belong to the class of SIMD architectures (Single-Instruction-Multiple-Data) [PH04]. At this point it should be mentioned, that NVIDIA, one of the main GPU manufacturers, proposes to overcome the SIMD model by the introduction of a SIMT model (Single-Instruction-Multiple-Thread) [NVI09], which may be considered as an extension to the SIMD model

that handles conditionals somewhat differently. The SIMT model provides the capability to switch between threads quickly, such that all processors follow the same path in the conditional and no core is disabled [HK12], [Lan09], [SZ10]. Still, the basic concept of the graphics pipeline remains, and also has implications on the memory concept of GPUs. While CPUs deal with memory bandwidth and latency limitations by using ever-larger hierarchies of caches, the number of cores in GPUs have grown approximately as fast as transistor density [Göd10]. Therefore, it is difficult to provide a large enough caching hierarchy on the GPU chip that delivers a reasonably high cache hit rate and maintains coherency. For this reason, the GPU caches are relatively small and the main focus concerning the design of the memory subsystem is put on maximizing the streaming bandwidth (and hence, throughput) by latency tolerance, page-locality, minimization of read-write direction changes and even lossless compression [Göd10].

Due to the hardware design and the concept of a graphics pipeline, using GPUs in scientific computing is especially interesting when targeting applications where the same operations are applied to a large set of data. While Owens et al. [OLG+07] provide a number of examples where significant performance increase could be achieved when porting inherently parallel algorithms to graphics processing units, Garland et al. [GLGN+08] provide a list of applications using NVIDIA's CUDA, an extension to the C programming language [NVI09]. In this programming paradigm, a so-called CUDA program consists of sequential C code that is executed by the host, and functions (called kernels) that process a large data set in parallel by the GPU. At the GPU level, the same kernel is executed in parallel by thousands or even millions of GPU threads. The GPU organizes the threads into a grid of several 1D, 2D or 3D thread blocks that are distributed among the multiprocessors [Göd10]. Each multiprocessor then executes one or multiple thread blocks in SIMD fashion and, in turn, each core of a multiprocessor runs one or more threads within a block in SIMT mode. The distinct multiprocessors handle the threads of the thread blocks in several small groups of threads called *warps*, where each warp contains 32 threads of 32 consecutive and increasing IDs and the first warp is the owner of thread 0. The threads of a warp execute concurrently the same instruction of a kernel but operate on different data and are free to follow the same or different execution paths without any synchronization point [OLG+07]. While the different threads within the same thread block execute the same instruction on different data and can communicate among themselves through barrier synchronization and the shared memory, there is no synchronization between the thread blocks of the grid, except that they read/write the input/output data from/to the global memory [OLG+07].

This concept of data partitioning into thread blocks resolves the problem of how to bring the necessary data to the computing cores in time, that occurs since the operation execution itself is usually very fast compared to the memory access. Each thread block is asynchronously streamed to the computing units, such that the data for the next operations is already communicated to the computing cores while the operations are executed on the current data. While this reduces waiting time and accelerates the overall performance [ABD+09], the order in which the thread blocks are processed is not determined. This is the fundamental principle of the block-asynchronous iteration we introduce in this chapter, which is also based on the efficient partitioning of the data into thread blocks that are then processed without deterministic order.

Conclusively, we want to mention that the market volume of interactive computer games amounts to billions of dollars per year, which leads to strong efforts in the hardware development of graphics processing units. Since the gamers' market is focused on performance, not on reliability or scientific computing features, the consumer line GPUs have a considerably higher number of bit-flips compared to general purpose CPUs and were not always IEEE compliant concerning precision formats and rounding from the beginning on (see

Appendix A). To meet the requirements of the scientific computing society using GPUs for general purpose computing, manufacturers have started to offer a more professional line of GPUs that fulfill higher reliability demands and often feature (larger) self-correcting memory called *ECC-memory* [NVI09].

4.2. Block-Asynchronous Iteration on GPUs

The asynchronous iteration method for GPUs that we propose can be seen as a two-stage iteration [ATDH12a]. This is due to the design of graphics processing units and the CUDA programming language [NVI09].
The idea is to split the linear system of equations into blocks of rows, and to assign the computations for each block to one thread block on the GPU [ATDH12a]. For these thread blocks, an asynchronous iteration method is used, while on each thread block, a synchronous Jacobi-like iteration method is performed. We denote this first version of the algorithm by *async-(1)*, referring that on each subdomain, one Jacobi iteration is conducted.
In a second step, we extend this basic algorithm to a version where the threads in a thread block perform multiple Jacobi iterations within the block. During the local iterations the x values used from outside the block are kept constant (equal to their values at the beginning of the local iterations). After the local iterations, the updated values are communicated. While this approach fits into the framework of asynchronous two-stage methods (see Section 3.3), the motivation can also be obtained from the well know hybrid relaxation schemes [BFKMY11, BFG$^+$]. The obtained algorithm is the block-asynchronous method we already introduced in Section 3.3.3, and its convergence is covered by Theorem 3.2.3, since it occurs as special case of the general asynchronous iteration for a specific update pattern. In domain-decomposition terminology, the blocks would correspond to subdomains, and thus we additionally iterate locally on every subdomain. We denote this scheme by *async-(local_iters)*, where *local_iters* denotes the number of Jacobi updates on every subdomain.

Another motivation for this approach comes from the hardware side. Especially the fact that the additional iterations almost come for free (as the subdomains are relatively small and the data needed largely fits into the multiprocessor's cache) motivates for adding local iterations on the subdomains. The obtained algorithm, visualized in Figure 4.1, can be written as component wise update of the solution approximation:

$$x_i^{(k+1)} += \frac{1}{a_{i,i}} \left(b_i \underbrace{- \sum_{j=1}^{T_S-1} a_{i,j} x_j^{(m-s(k+1,j))}}_{\text{global part}} \underbrace{- \sum_{j=T_S}^{T_E} a_{i,j} x_j^{(m)}}_{\text{local part}} \underbrace{- \sum_{j=T_E+1}^{n} a_{i,j} x_j^{(k-s(k+1,j))}}_{\text{global part}} \right), \tag{4.1}$$

where T_S and T_E denote the starting and the ending indices of the matrix/vector part in the thread block. Furthermore, for the local components, the always newest values are used, while for the global part, the values from the beginning of the iteration are used [ATDH12a]. The shift function $s(k+1, j)$ denotes the iteration shift for the component j which can be positive or negative, depending on whether the thread block where the component j is located in already has conducted more or less iterations. Note that this may give a block Gauss-Seidel flavor to the updates.

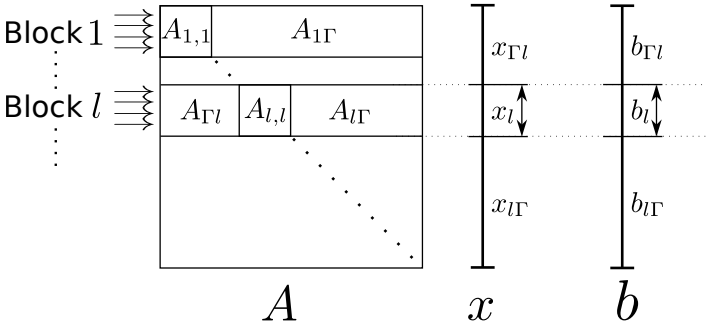


Figure 4.1.: Visualizing the asynchronous iteration in block description used for the GPU implementation. $A_{l,l}$ denotes the l-th diagonal block, $A_{\Gamma l}$ and $A_{l\Gamma}$ the block left, respectively right, of $A_{l,l}$. Consistent notation is used for the block decomposition of the vectors.

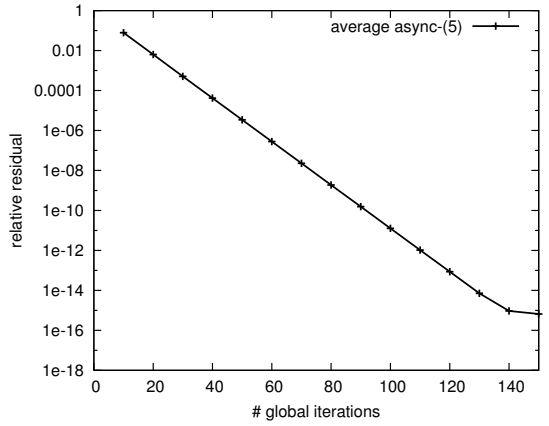
4.3. Experiments on the Non-deterministic Behavior of Asynchronous Iteration

As already mentioned, asynchronous iteration methods are by definition non-deterministic [CM69]. While synchronous relaxation algorithms always provide the same solution approximations for each solver run, the unique pattern concerning the distinct component updates in their asynchronous counterparts generates a sequence of iteration approximations, that can usually only be reproduced by choosing exactly the same update order. This also implies that variations in the convergence rate may occur for the individual solver runs. If the scheduling for the individual component updates is based on a recurring pattern, these variations in the convergence behavior may increase with the number of iterations since the component update pattern may multiply its influence for higher iteration counts. When iterating locally in the block-asynchronous approach (see Section 4.2), the influence of the unique component update order potentially has even more impact since the local iterations do not account for the off-block entries. This may especially become a critical issue when targeting linear systems with considerable off-diagonal parts.

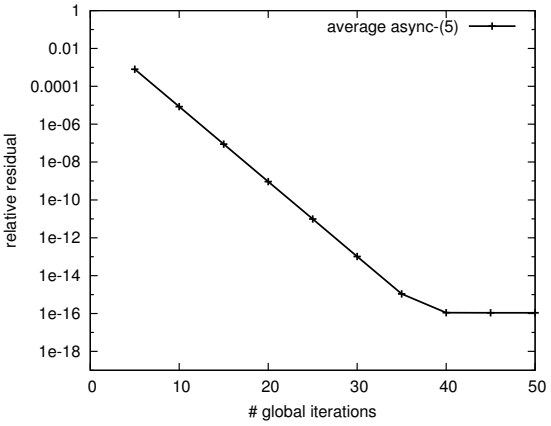
To investigate the issue of the non-deterministic behavior, we conduct multiple solver runs using the same experiment setup and monitor the relative residual behavior for the different approximation sequences. As the main focus of this chapter is on the block-asynchronous method, we apply the async-(5) algorithm based on a moderate block size of 128, which allows for a strong influence of the non-deterministic GPU-internal scheduling of the threads [NVI09]. Targeting the matrices FV1 and TREFETHEN_2000 ensures, that we include results for very different matrix structures, i.e. these systems have very different diagonal dominance (see Appendix B). All tests on the non-deterministic behavior are based on 1000 solver runs using the same hardware and software configuration on the GPU-accelerated Supermicro system (see Appendix C.1 for details about the hardware).

For each matrix system, we report in Figure 4.2 the variations in convergence as difference between the largest and smallest relative residual in absolute and relative values (relative to the average residual). The exact numbers for these plots can be found in Table 4.1 and 4.2. Additionally, we provide in Table 4.3 and 4.4 information about the statistical parameters variance, standard error and standard deviation [Hen10].

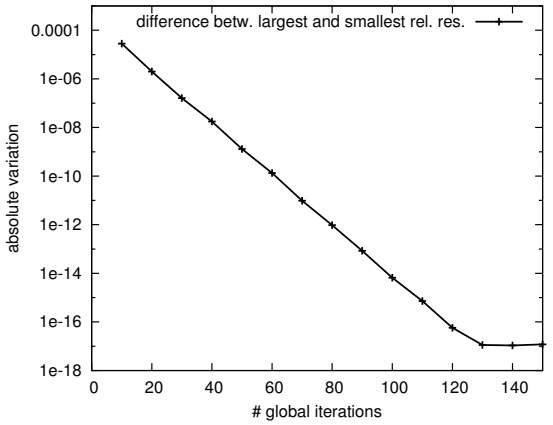
From Figure 4.2a we can deduce that the async-(5) method using the given experiment setup converges within about 130 global iterations for matrix FV1. Having achieved convergence, also the absolute variations in between the fastest and slowest convergence approach a limit, see Figure 4.2c. While the absolute values provide an general idea about the magnitude of the variations, investigating the relative variances in Figure 4.2e allows for a



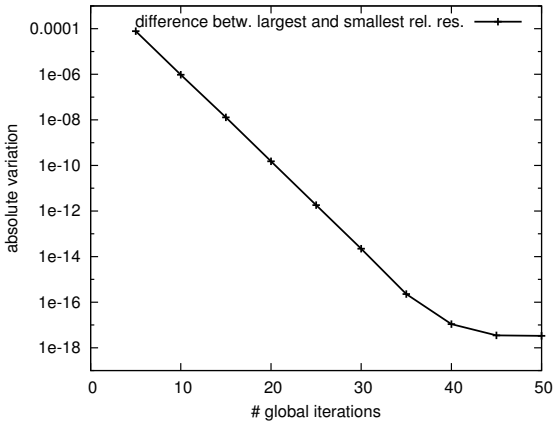
(a) average convergence, FV1



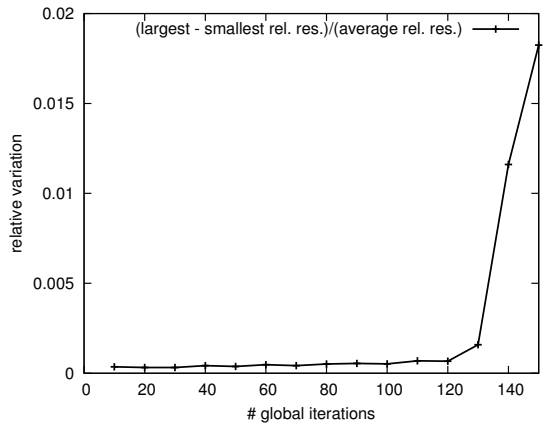
(b) average convergence, TREFETHEN_2000



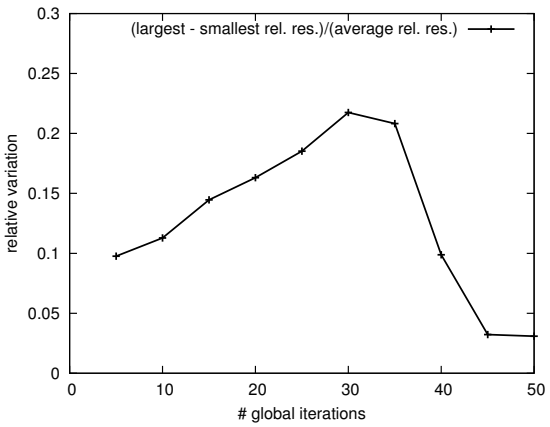
(c) absolut variations, FV1



(d) absolut variations, TREFETHEN_2000



(e) relative variations, FV1



(f) relative variations, TREFETHEN_2000

Figure 4.2.: Visualizing the average convergence, the absolute respectively relative variations in the convergence behavior of async-(5) depending on the number of conducted global iterations.

more efficient quantification of the differences. Obviously, there exist some variations, but they are very small, and may even be neglected. The large relative variations for >130 iterations can be explained by rounding effects when computing the relative variation using the very small values (see Table 4.2) on a system with limited accuracy. The overall small variations were expected due to the design and sparsity pattern of the matrix: FV1 is symmetric and almost all elements are gathered on the diagonal blocks, and therefore are accounted for in the local iterations. Similar tests on diagonal dominant systems with the same sparsity pattern but higher condition number reveal, that the latter one has only small impact on the variations between the individual solver runs, see [ATDH12a].

For the system TREFETHEN_2000 we converge within about 40 iterations (see Figure 4.2b). Only very few solver runs have not reached convergence after 45 iterations. Like in the FV1 case, we observe in Figure 4.2d the expected exponential decrease of the absolute variation. Concerning the relative variations, the results look quite different: All values for the relative variations are significantly higher than for the test matrix FV1. Close to convergence, the relative difference between largest and smallest relative residual approximates 20 %. This confirms the expectation, that the larger off-block parts in TREFETHEN_2000 emphasize the non-deterministic GPU-internal scheduling since they are not accounted for in the local iteration on the subdomains (see Section 4.2 for the algorithm design). Furthermore, we can identify a dependency between iteration count and the relative variation: The component update orders multiply their impact when conducting a high number of iterations, and the relative difference between the individual approximations rises (see Figure 4.2f). The linear growth of the relative variations immediately suggests the existence of a recurring pattern in the GPU-internal scheduling, which amplifies the variations in the convergence of the different solver runs.
In Table 4.3 and 4.4 we additionally provide information about statistical parameters like variance, standard deviation and standard error.

Summarizing the results, we can conclude that the scheduling of the threads has influence on the convergence behavior of the block-asynchronous iteration. Especially for systems with significant off-diagonal parts, the effects should be considered when using the method in scientific computing. For diagonal dominant systems on the other hand, the variations may be negligible. Furthermore, it seems that the GPU-internal scheduling is based on a recurring pattern which ensures that at some point of the iteration run, all components have been updated similarly often. Due to this pattern, it may be useful to apply larger block-sizes that allow for less possibilities in the scheduling. Larger block-sizes usually come along with the advantage of accounting for more elements in the local iterations, and therewith faster convergence. Only for very specific matrix properties it may be reasonable to choose small block sizes, i.e. if the off-diagonal parts can be reduced or the matrix size is a multitude of the smaller block size, see Section 4.10.6.) To account for this finding we will in general stick to large block sizes for all further experiments. However, the main consequence of the results is a convention for the rest of the thesis:

Although we will not stress it explicitly every time, all further results should be considered as average using several solver runs.

4.4. Experiments on Block-Asynchronous Iteration on GPUs

The experimental results presented in the following sections are also part of the conference contribution [ATDH12a].

4.4.1. Convergence rate of Asynchronous Iteration on GPUs

In the first experiment, we analyze the convergence behavior of the async-(1) iteration method and compare it with the convergence rate of the Gauss-Seidel and Jacobi method.

# iters	averg. residual	max. res	min. res.	abs. variation	rel. variation
10	7.8304e-02	7.8313e-02	7.8285e-02	2.8111e-05	3.5885e-04
20	6.3243e-03	6.3251e-03	6.3231e-03	2.0125e-06	3.1782e-04
30	5.1392e-04	5.1400e-04	5.1383e-04	1.6310e-07	3.1716e-04
40	4.1902e-05	4.1912e-05	4.1895e-05	1.7607e-08	4.2002e-04
50	3.4238e-06	3.4244e-06	3.4231e-06	1.3232e-09	3.7969e-04
60	2.8019e-07	2.8026e-07	2.8012e-07	1.3318e-10	4.7467e-04
70	2.2956e-08	2.2961e-08	2.2951e-08	9.7834e-12	4.2254e-04
80	1.8825e-09	1.8829e-09	1.8820e-09	9.6127e-13	5.0995e-04
90	1.5449e-10	1.5453e-10	1.5445e-10	8.4035e-14	5.4372e-04
100	1.2685e-11	1.2689e-11	1.2682e-11	6.6530e-15	5.2025e-04
110	1.0422e-12	1.0426e-12	1.0418e-12	7.2436e-16	6.9082e-04
120	8.5725e-14	8.5753e-14	8.5695e-14	5.7262e-17	6.6724e-04
130	7.1399e-15	7.1458e-15	7.1345e-15	1.1334e-17	1.5823e-03
140	9.2748e-16	9.3342e-16	9.2265e-16	1.0767e-17	1.1608e-02
150	6.5579e-16	6.6219e-16	6.5022e-16	1.1964e-17	1.8243e-02

Table 4.1.: Variations in the convergence behavior for 1000 solver runs on FV1. The iteration number is the global iteration count.

# iters	averg. residual	max. res	min. res.	abs. variation	rel. variation
5	8.0190e-04	8.1516e-04	7.3689e-04	7.8277e-05	9.7614e-03
10	8.4330e-06	8.6821e-06	7.7307e-06	9.5147e-07	1.1282e-01
15	8.8600e-08	9.2472e-08	7.9658e-08	1.2813e-08	1.4462e-01
20	9.3022e-10	9.8491e-10	8.3319e-10	1.5171e-10	1.6309e-01
25	9.7817e-12	1.0427e-11	8.6158e-12	1.8120e-12	1.8524e-01
30	1.0260e-13	1.1038e-13	8.8065e-14	2.2314e-14	2.1747e-01
35	1.0906e-15	1.1960e-15	9.6899e-16	2.2705e-16	2.0818e-01
40	1.1012e-16	1.1843e-16	1.0755e-16	1.0881e-17	0.9880e-01
45	1.0811e-16	1.1041e-16	1.0692e-16	3.4880e-18	3.2261e-02
50	1.0811e-16	1.1057e-16	1.0723e-16	3.3381e-18	3.0875e-02

Table 4.2.: Variations in the convergence behavior for 1000 solver runs on TREFETHEN_2000. The iteration number is the global iteration count.

# iters	variance	standard deviation	standard error
10	1.6334e-11	4.0415e-06	2.0233e-07
20	1.1244e-13	3.3532e-07	1.6787e-08
30	7.4425e-16	2.7281e-08	1.3657e-09
40	6.7713e-18	2.6021e-09	1.3027e-10
50	4.9401e-20	2.2226e-10	1.1127e-11
60	4.1092e-22	2.0271e-11	1.0148e-12
70	2.9781e-24	1.7257e-12	8.6395e-14
80	2.6892e-26	1.6398e-13	8.2096e-15
90	1.9368e-28	1.3916e-14	6.9672e-16
100	1.4737e-30	1.2139e-15	6.0775e-17
110	1.2855e-32	1.1338e-16	5.6761e-18
120	8.8288e-35	9.3962e-18	4.7039e-19
130	4.6302e-36	2.1518e-18	1.0772e-19
140	3.1601e-36	1.7776e-18	8.8994e-20
150	3.9031e-36	1.9756e-18	9.8905e-20

Table 4.3.: Statistics in the convergence behavior for 1000 solver runs on FV1. The iteration number is the global iteration count.

# iters	variance	standard deviation	standard error
5	1.6769e-10	1.2949e-05	4.0970e-07
10	2.8159e-14	1.6780e-07	5.3091e-09
15	4.8964e-18	2.2127e-09	7.0009e-11
20	7.0058e-22	2.6468e-11	8.3742e-13
25	9.4666e-26	3.0767e-13	9.7345e-15
30	1.2095e-29	3.4778e-15	1.1003e-16
35	1.5321e-33	3.9142e-17	1.2384e-18
40	2.4862e-36	1.5767e-18	4.9887e-20
45	2.2543e-37	4.7479e-19	1.5021e-20
50	2.2468e-37	4.7400e-19	1.4996e-20

Table 4.4.: Statistics in the convergence behavior for 1000 solver runs on TREFETHEN_2000. The iteration number is the global iteration count.

Note that the residuals in all reported data of this and the following sections are always in the L^2 norm ($\| r \|_2$).

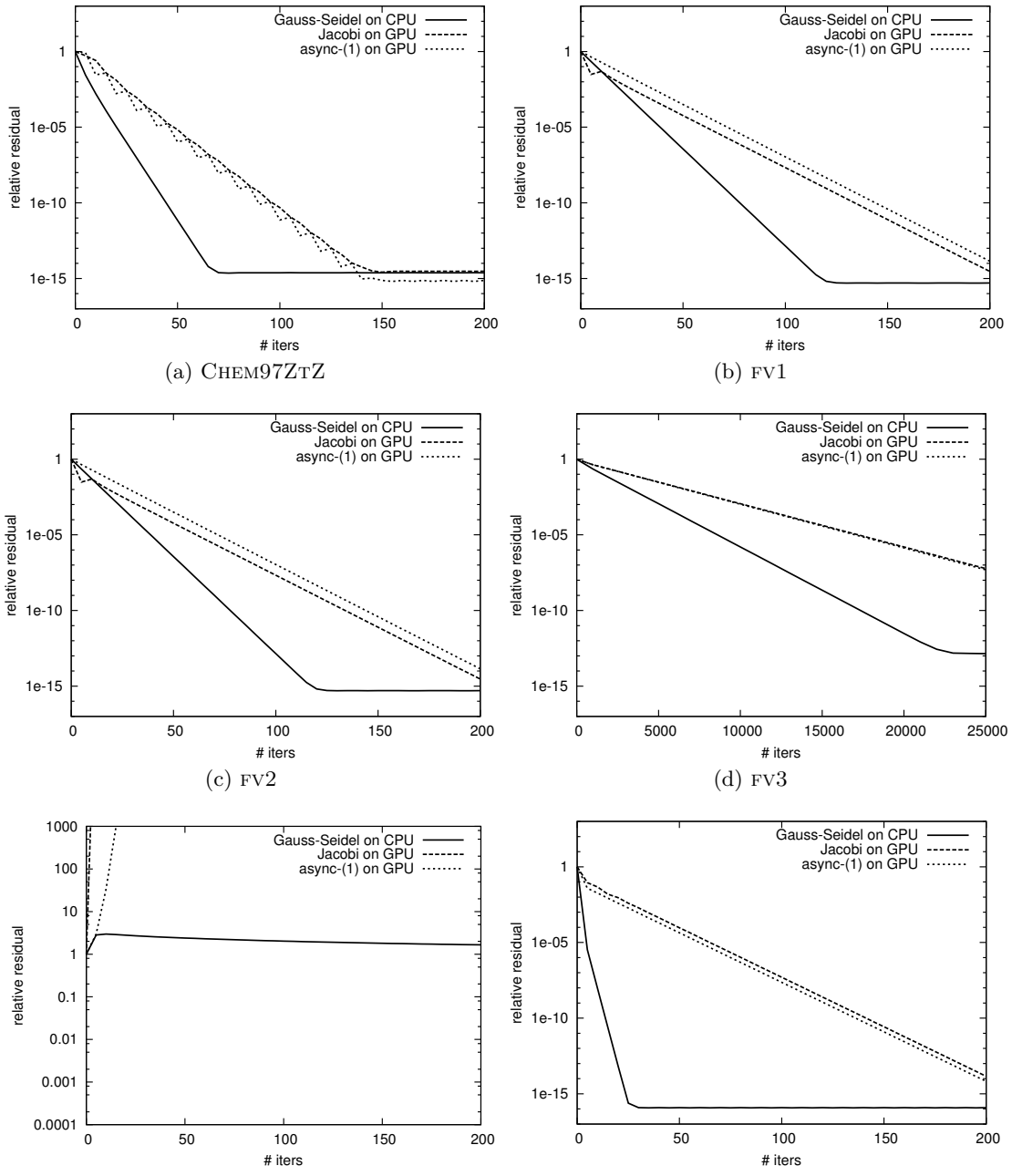
The experiment results, summarized in Figure 4.3, show that for test systems CHEM97ZTZ, FV1, FV2, FV3 and TREFETHEN_2000 the synchronous Gauss-Seidel algorithm converges in considerably less iterations than the asynchronous iteration. This superior convergence behavior is intuitively expected, since the synchronization after each component update allows for the usage of the updated components immediately for the next update in the same global iteration. For the Jacobi implementation, the synchronization after each iteration still ensures the usage of all updated components in the next iteration. Since this is not true for the asynchronous iteration, the convergence depends on the problem and the update order. While the usage of updated components implies the potential of a Gauss-Seidel convergence rate, the chaotic properties may trigger convergence slower than Jacobi.

Still, we observe for all test cases convergence rates similar to the synchronized Jacobi iteration, which is still low compared to Gauss-Seidel.

The results for test matrix S1RMT3M1 (Figure 4.3e) show an example where neither of the methods is suitable for direct use. The reason is that here $\rho(B) > 1$ (in particular, $\rho(B) \approx 2.65$, see Table B.1 in the Appendix). Nevertheless, note that this matrix is symmetric and positive definite (SPD) and Jacobi-based methods still can be used after a proper scaling is added, e.g., taking $B = I - \tau D^{-1}A$ with $\tau = \frac{2}{\lambda_1+\lambda_n}$, where λ_1 and λ_n approximate the smallest and largest eigenvalue of $D^{-1}A$.

4.4.2. Convergence rate of Block-Asynchronous Iteration

Like in [ATDH12a] we now consider the block-asynchronous iteration method introduced in Section 4.2, which additionally performs a set of Jacobi-like iterations on every subdomain. In Table 4.5 we report the overhead triggered by the additional local iterations conducted on the subdomains. Switching from async-(1) to async-(2) affects the total computation time by less than 5%, independent of the total number of global iterations. At the same time, this leads to an algorithm where every component is updated twice as often. Even if we iterate every component locally by 9 Jacobi iterations, the overhead is less than 35%, while the total updates for every component differ by a factor of 9 [ATDH12a]. There exists though a critical point, where adding more local iterations does not improve the overall performance. It is difficult to analyze the trade-off between local and global iterations [MY11], and we desist from giving a general statement for the optimal choice of local iterations. This is due to the fact that the choice depends not only on the characteristics of the linear problem, but also on the iteration status of the thread block and the local components (as related to the asynchronism), subdomain sizes, and other parameters. Based on empirical tuning and practical experience (trying to match the convergence of the new method to that of a Gauss-Seidel iteration) we set the number of local Jacobi-like updates to five. Therefore we choose async-(5) for all subsequent analysis of the block-asynchronous iteration method in this chapter. The additional local iterations in asynchronous methods provide less contribution to the iteration process than global ones in synchronized algorithms, as they do not take into account off-block entries. We note that as a consequence, the number of iterations in asynchronous algorithms can not directly be compared with the number of iterations in a synchronized algorithm. To account for this mismatch and the fact that the local iterations almost come for free in terms of computational effort, we from now on use the convention of counting only the number of of global iterations, where every single component is updated five times as often by iterating locally.



(a) Chem97ZtZ (b) fv1

(c) fv2 (d) fv3

(e) s1rmt3m1 (f) Trefethen_2000

Figure 4.3.: Convergence behavior for different test matrices. Relative residuals in L^2 norm.

method	computation time [sec] for # global iterations 100	200	300	400	500
async-(1)	1.376425	2.437521	3.501462	4.563519	5.624792
async-(2)	1.431110	2.546361	3.660030	4.773864	5.891870
async-(3)	1.482574	2.654470	3.819478	4.987472	6.156434
async-(4)	1.532940	2.749808	3.972644	5.191812	6.410378
async-(5)	1.577105	2.838185	4.099068	5.363081	6.655686
async-(6)	1.629628	2.938897	4.255335	5.569045	6.879329
async-(7)	1.680975	3.044979	4.412199	5.778823	7.144304
async-(8)	1.736295	3.148895	4.571684	5.990520	7.409536
async-(9)	1.786658	3.259132	4.730689	6.202893	7.676786

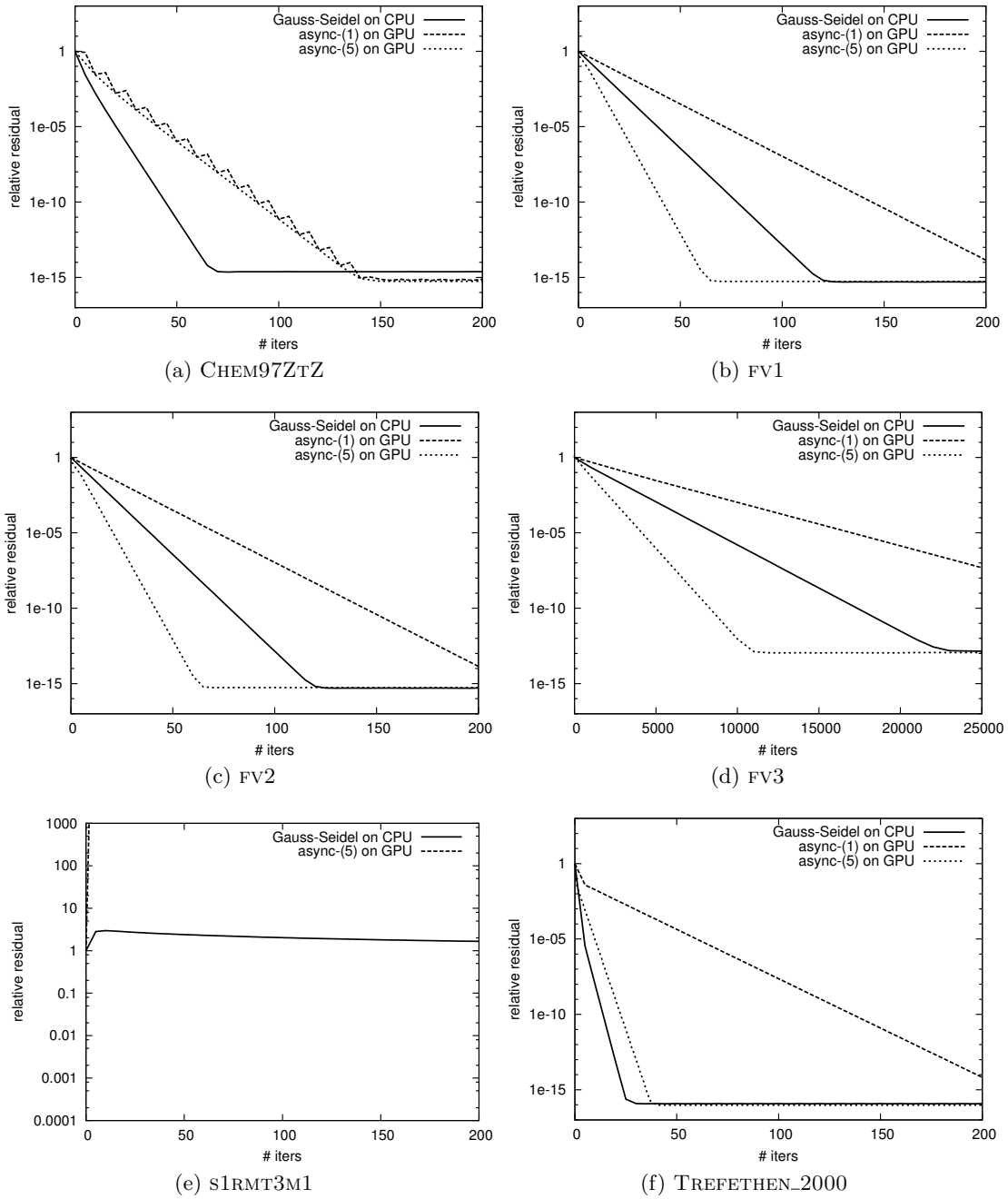
Table 4.5.: Overhead to total execution time by adding local iterations, matrix FV3.

Using this notation we now aim for comparing in Figure 4.4 the convergence rate of async-(5) with the Gauss-Seidel convergence rate.

As theoretically expected, synchronous relaxation as well as the block-asynchronous async-(5) are not directly suitable to use for the S1RMT3M1 matrix. Besides this case, the async-(5) improves the convergence rate of async-(1) for all other test cases. While, depending on the matrix structure, we may expect an improvement factor of up to five, in the experiments we observe improvements of up to four.

The rule of thumb expectation for the convergence rate of the async-(5) algorithm is based on the rate with which values are updated and the rate of propagation for the updates. For example, this is the observation that Gauss-Seidel often converges about twice as fast as Jacobi [KK03]. In other words, four Jacobi iterations would be expected (in general) to provide residual reduction approximating two Gauss-Seidel iterations. The experiments show that the convergence of async-(5) for CHEM97ZTZ is characteristic for the convergence of the synchronous Jacobi iteration. This can be explained by the fact that the local matrices for CHEM97ZTZ are diagonal and therefore it does not matter how many local iterations would be performed. An improvement for this case could potentially be obtained by reordering. The case for TREFETHEN_2000 is similar: although there is improvement compared to Jacobi, the rate of convergence for async-(5) is not twice as fast as Gauss-Seidel, and the reason is again the structure of the local matrices (see Figure B.3 in the Appendix for the structure of the matrices and Figures 4.4a and 4.4f for the convergence results). Considering the remaining linear systems of equations FV1, FV2 and FV3, we obtain approximately twice as fast convergence when replacing Gauss-Seidel by the async-(5) algorithm (see Figures 4.4b, 4.4c, and 4.4d). Since for these cases most of the relevant matrix entries are gathered on or near the diagonal and therefore are taken into account in the local iterations on the subdomains, we observe significant convergence gain when iterating locally. Hence, as long as the asynchronous method converges and the off-block entries are "small", adding local iterations may be used to not only compensate for the convergence loss due to the chaotic behavior, but moreover to gain significant overall convergence improvements [ATDH12a].

But the convergence rate alone does not determine whether an iterative method is efficient or not. The second important metric we have to consider is the time needed to process one iteration on the respective hardware platform. While the time can be easily measured for the synchronous iteration methods, the nature of asynchronous relaxation schemes does not allow the straight forward determination of the time needed per iteration, since not all components are updated at the same time. Especially, it may happen that some blocks were already iterated several times, while other blocks were not processed at all. For this



(a) Chem97ZtZ (b) fv1

(c) fv2 (d) fv3

(e) s1rmt3m1 (f) Trefethen_2000

Figure 4.4.: Convergence rate of block-asynchronous iteration. The relative residual is in L^2 norm, the iteration count denotes in the async-(5) case the number of global iterations.

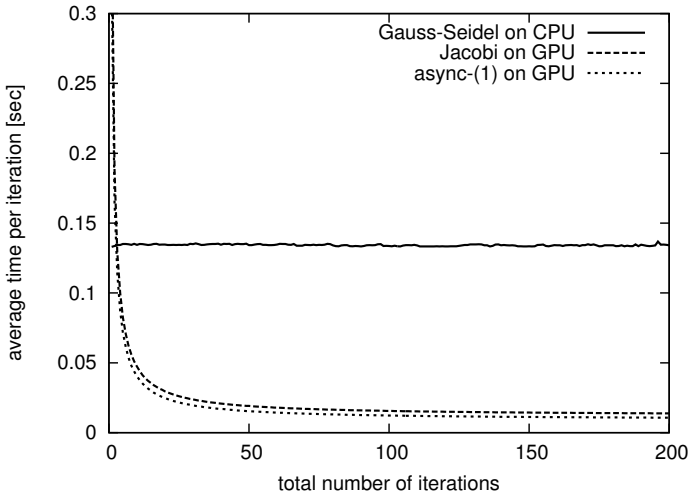


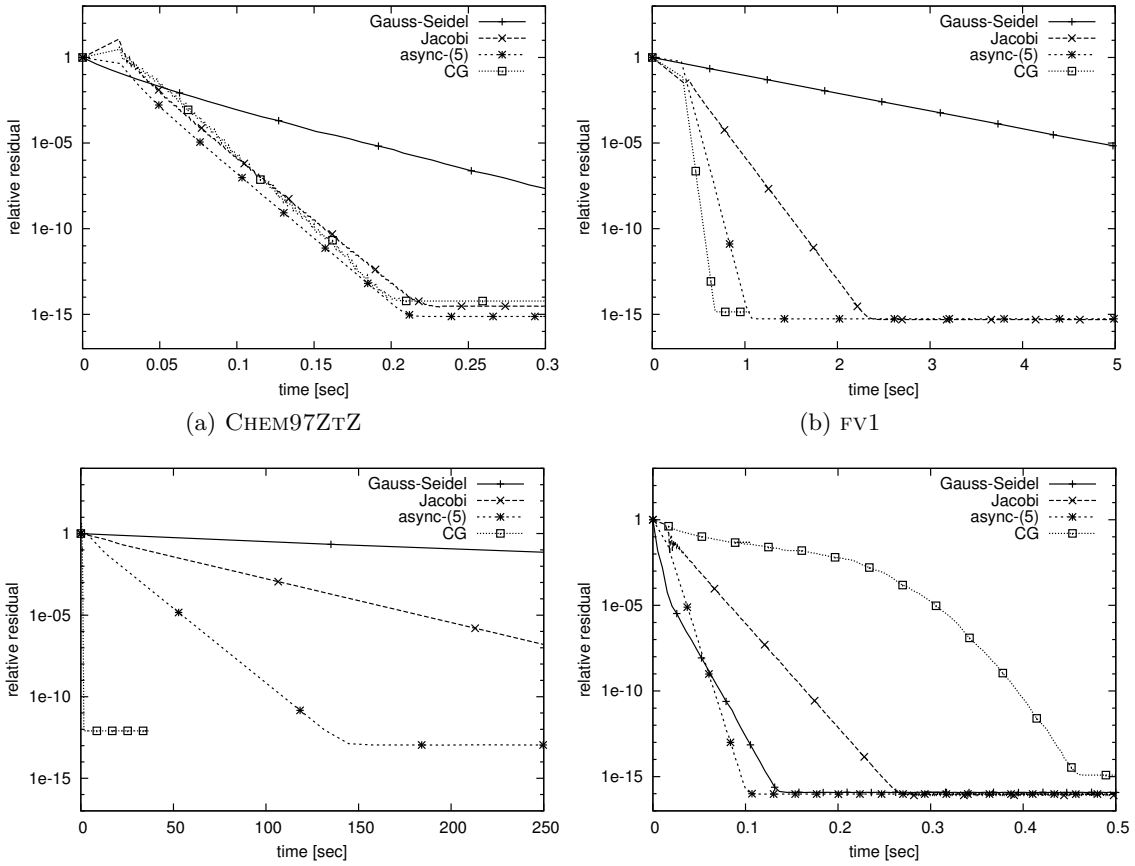
Figure 4.5.: Average iteration timings of CPU/GPU implementations depending on total iteration number, test matrix FV3.

Matrix name	G.-S. (CPU)	Jacobi (GPU)	async-(5) (GPU)
CHEM97ZTZ	0.008448	0.002051	0.001742
FV1	0.120191	0.019449	0.012964
FV2	0.125572	0.020997	0.014729
FV3	0.125577	0.021009	0.014737
S1RMT3M1	0.039530	0.006442	0.004967
TREFETHEN_2000	0.007603	0.001494	0.001305

Table 4.6.: Average iteration timings in seconds per global iteration.

reason, only an average time per global iteration can be computed by dividing the total time by the total number of iterations. Therefore, we also use an average time for the CPU implementation. It should also be mentioned that, while the average timings for one iteration on the CPU are almost constant, for the GPU implementations the iteration time differs considerably. This stems from the fact, that all timings include the data transfers between host and GPU. Hence, we have considerable communication overhead when performing only a small number of iterations, while the average computation time per iteration decreases significantly for cases where a large number of iterations is conducted. This behavior is shown in Figure 4.5, where the average iteration timings for the test matrix FV3 are reported. For the other test matrices, the average timings for the Gauss-Seidel implementation on the CPU and the Jacobi and async-(5) iteration on the GPU are shown in Table 4.6 where we took the average of the cases when conducting $10, 20, 30 \ldots 200$ iterations for the GPU implementations. Note that the iteration time for Jacobi is, due to the synchronization after each iteration, higher than for async-(5), despite the five local updates in the asynchronous method.

Overall, we observe, that the average iteration time for the async-(5) method using the GPU is only a fraction of the time needed to conduct one iteration of the synchronous Gauss-Seidel on the CPU. While for small iteration numbers and problem sizes we have a factor of around 5, it rises to over 10 for large systems and high total iteration numbers. The question is, whether the faster component updates can compensate for the slower convergence rate when targeting the matrices CHEM97ZTZ and TREFETHEN_2000. In this case, the block-asynchronous method using the GPU as accelerator would still outperform the synchronous Gauss-Seidel on the CPU.



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Figure 4.6.: Relative residual behavior with respect to solver runtime.

4.4.3. Performance of the Block-Asynchronous Iteration Method

While the convergence rate analyzed in Section 4.4.1 and 4.4.2 is especially interesting from the theoretical point of view, the more relevant metric when using iterative methods to solve a linear system of equations is the time needed to provide a solution approximation of a certain accuracy. This depends not only on the convergence rate, but also on the hardware- and implementation-specific iteration rate. In [ATDH12a] the block asynchronous iteration performance is compared to the performance of Gauss-Seidel and Jacobi. Now, we extend the analysis by comparing also to a highly tuned GPU implementation of the CG solver (see Section 2.5.3). Also, instead of reporting only time-to-accuracy, we show the residual decrease over the computing time. This enables to determine the optimal solver also for cases where low accuracy approximations are sufficient. Such scenarios occur for example in the solution process of nonlinear equations where only coarse solutions for the first linearization steps are required.

Due to the results in Figure 4.4 it is reasonable to limit the performance analysis to the matrices Chem97ZtZ, fv1, fv3 and Trefethen_2000: The matrix characteristics and convergence results for fv2 are very similar to fv1, and for the s1rmt3m1 problem we observed that neither Gauss-Seidel nor Jacobi or block-asynchronous iteration are suitable methods.

In Figure 4.6 the time-dependent residuals for the different solver implementations are reported (relative residuals in L^2-norm).

For the very diagonally dominant systems fv1 and fv3 the performance improvement when switching from Jacobi to block-asynchronous iteration are significant: The async-(5)

method converges (with respect to computation time) almost twice as fast as Jacobi. At the same time, both methods outperform the sequential (CPU-based) Gauss-Seidel solver by orders of magnitude. Still, the performance of the CG method can not be achieved. This was expected, since the CG method belongs to the most efficient iterative solvers for symmetric positive definite systems. In Figure 4.6b async-(5) converges about twice as fast as Jacobi, significantly faster than Gauss-Seidel, but the CG method still shows about one-third higher performance. For the system FV3 with considerably higher condition number, the differences are even more significant (see Figure 4.6c): the time-to solution needed by the synchronized CG is only a fraction of the computation time of Jacobi, async-(5) or Gauss-Seidel. Only if the approximation accuracy is relevant, applying async-(5) could be considered to post-iterate the solution approximation.

For strongly coupled problems, which result in matrix systems containing large off-diagonal parts, the performance differences between Jacobi and block-asynchronous iteration decrease. This stems from the fact that the entries located outside the subdomains are not taken into account for the local iterations in async-(5) [ATDH12a]. Thus, it is expected that for the problem CHEM97ZTZ the performance results for Jacobi and block-asynchronous iteration are very similar. They are also almost equal to the performance of the, algorithmically very different, CG solver. Only the CPU-based Gauss-Seidel converges slower with respect to computation time (see Figure 4.6a). Concerning the three superior methods, the block-asynchronous iteration outperforms not only the Jacobi method, but even the highly optimized CG solver.

Probably the most interesting results occur for the TREFETHEN_2000 problem (see Figure 4.6d). The async-(5) method using the GPU does not reach the Gauss-Seidel performance on the CPU for small iteration numbers, which may be caused by the characteristics of the problem: As the linear system combines small dimension with low condition number, both enabling fast convergence, the overhead triggered by the GPU kernel calls is for small iteration numbers crucial [ATDH12a]. Going to higher iteration numbers, the async-(5) outperforms the CPU implementation of Gauss-Seidel also for this matrix. Compared to CG and Jacobi, the async-(5) method is superior for any approximation accuracy.

We conclude from the analysis in this section, that asynchronous iteration schemes using parallel devices have to be used carefully, but for suitable systems of linear equations and appropriate hardware platforms we may expect significant performance advantages when replacing the synchronous counterparts [ATDH12a].

4.5. Fault-Tolerance of Block-Asynchronous Iteration

This section is dedicated to analyze the block-asynchronous iteration introduced in Section 4.2 with respect to error resilience. Particularly, we want to analyze how hardware failure impacts the method's convergence and performance characteristics. The topic of fault resilient algorithms is of significant importance, as we expect that the high complexity in future hardware systems comes along with a high failure rate [CGG+09, PBGM09].
Current high performance parallel systems consist of about one million processing elements, but aiming for Exascale computing, this number has to be increased by about three orders of magnitude [BBC+08]. This increase in component number is expected to be faster than the increase in component reliability, with projections in the minutes or seconds for Exascale systems. From the current knowledge and observations of existing large systems, it is anticipated that especially a rise of the *Silent Errors* may take place. These are errors that get detected after long time having caused serious damage to the algorithm, or never get detected at all [ABC+10].
For most synchronized iterative solvers hardware failure is crucial, resulting in the breakdown of the algorithm. For implementations that do not feature checkpointing or recovery

techniques [BHF+12, HH11, MRK10, DBB+11, DLTD11, BDB+12, DBB+12], the complete solution process has to be restarted. While checkpointing strategies are widespread used in today's implementations, algorithms will no longer be able to rely on checkpointing to cope with faults in the Exascale era. This stems from the fact, that the time for checkpointing and restarting will exceed the mean time of failure of the full system [ABC+10]. Hence, a new fault will occur before the application could be restarted, causing the application to get stuck in a state of constantly being restarted. The nature of asynchronous methods removes the need for checkpointing: the high tolerance to update order and communication delay implies that, as long as all components are updated at some point, they are resilient to hardware failure. The inherent reliability with respect to detectable hardware failure makes asynchronous methods suitable candidates for exascale systems based on a high number of processing elements. In case of silent errors that do not get detected, also the asynchronous methods will usually not converge to the solution. This implies, that for problems where convergence is expected, a convergence delay or non-converging sequence of solution approximations indicates that a silent error has occurred. Especially a delay in the expected convergence may portend that a temporal hardware failure has taken place, e.g. the power-off of one component due to overheating. Hence, additionally to the fault-tolerance with respect to detectable hardware errors, it is also possible to imagine scenarios where asynchronous methods can be used to detect silent errors.

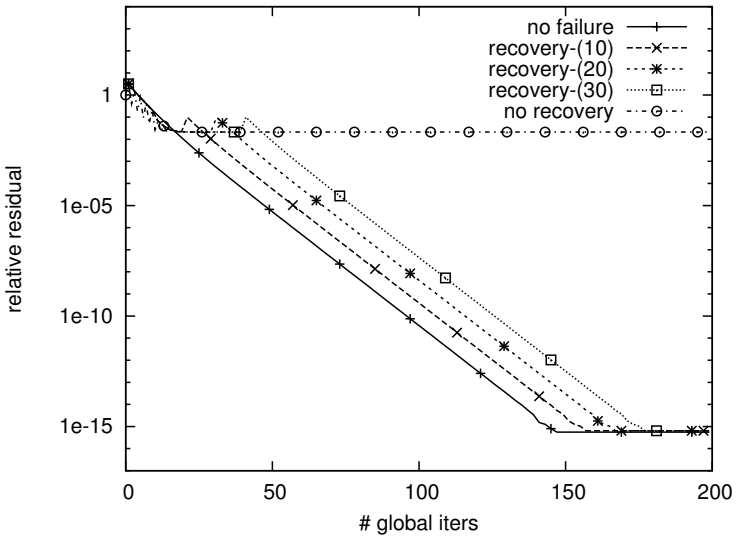
To investigate the issue of fault tolerance experimentally, we consider the following scenario:
Block-asynchronous iteration is used on a system with a high core number to solve a linear system of equations. At some point, a certain number of the cores iterating the distinct components break down. Within a certain time frame, the operating system detects the hardware failure and may reconfigure the algorithm during runtime by assigning the respective components to other (e.g. additional) cores or fix the corrupted ones.

This setup is realistic as it may occur due to several reasons. One example is the failure of computing units of the different layers (cores, CPUs, nodes), a common *hardware error* scenario [CGG+09]. Operating system software may detect hardware breakdown after some time, and it is a question of the implementation how these hard errors are handled.

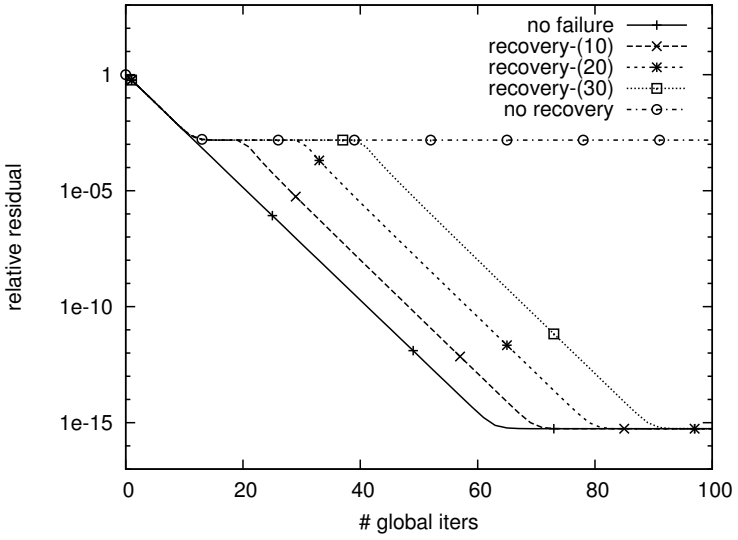
In our experiments we simulate the introduced scenario the following way:

At a certain time t_0, a preset number of randomly chosen components is no longer considered in the iteration process. We report the residual behavior of different implementations that either detect the failure and reassigns the components to other cores after a certain recovery time t_r, or do not recover at all. While we expect a delay in the convergence for the recovery case, the algorithms not reassigning the components handled by broken cores may generate a solution approximation with significant residual error. We consider the test matrices CHEM97ZTZ, FV1 and TREFETHEN_2000, that are due to their very different characteristics suitable candidates (see Appendix B). For all experiments, we simulate the hardware breakdown of 25% of the computing cores after about $t_0 = 10$ global iterations. This implies, that one fourth of the components (randomly chosen) is no longer updated. We implement different versions, where the algorithm detects and reassigns the components after $t_r = 10, 20, 30$ global iterations, or does not recover at all.

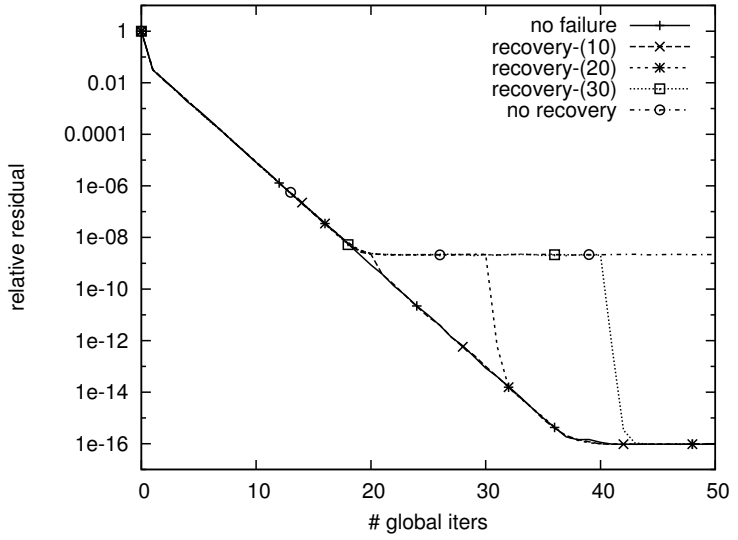
In Figure 4.7 we report the relative residual behavior for the different implementations and observe very similar results for all matrix systems. In the non-recovering case, the algorithm generates a solution approximation that differs significantly from the exact solution. Especially, continuing the iteration process for the remaining components (handled by the working cores) has no influence on the generated values for the symmetric positive definite test matrices. As soon as the iteration process recovers by assigning the workload



(a) Chem97ZtZ



(b) fv1



(c) Trefethen_2000

Figure 4.7.: Convergence of async-(5) for hardware failure. The implementations either recover by reassigning the components to other cores after the recovery time t_r (denoted with recover-(t_r)), or generate a solution approximation with significant residual error.

recovering async-(5)	recover-(10)	recover-(20)	recover-(30)
CHEM97ZTZ	8.22	13.67	20.45
FV1	8.16	19.50	31.66
TREFETHEN_2000	8.16	11.45	16.61

Table 4.7.: Additional computation time in % needed for the async-(5) featuring different recovery times t_r to provide the solution approximation.

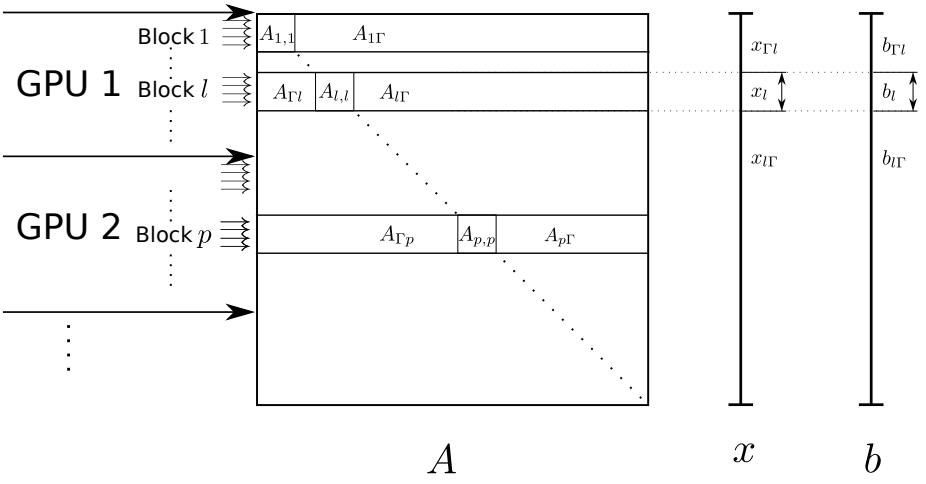


Figure 4.8.: Visualizing the block-asynchronous iteration for multi-GPU implementation.

of the broken cores to other hardware components, the convergence is retrieved. With some problem specific delay (see Table 4.7), the same solution approximation is generated like for the case of no error. This reveals the high resilience of block-asynchronous iteration with respect to hardware failure, and the high reliability of the methods.

4.6. Block-Asynchronous Iteration on Multi-GPU Systems

Using multiple graphics processing units, an additional layer of asynchronism is added to the block-asynchronous iteration method. Now we may split the original system into blocks that are addressed to the different devices, whereas the local blocks of rows are again split into even smaller blocks of rows, that are then assigned to the different thread blocks on the respective GPUs. In between the GPUs as well as in between the different thread blocks, an asynchronous iteration method is conducted, while on each thread block, multiple Jacobi-like iteration steps can be applied (see Figure 4.8). For this reason, one could consider the multi-GPU implementation of the block-asynchronous iteration also as three-stage iteration, but as both outer (block) updates are asynchronous, there is no algorithmic difference to the two-stage iteration. The inter-GPU and inter-block iterations are then considered as one asynchronous block-component update.

For implementing the asynchronous iteration on multi-GPU systems, the central question is how the updated values can be communicated efficiently. In the classical approach to multi-GPU implementations, the main memory serves as communication facility, transferring data to the different devices. This usually triggers a high CPU workload and, due to the PCI-connection, a bottleneck in the algorithm. CUDA 4.0 offers a whole set of possibilities, replacing the former slow communication via the host [NVI11]. While the asynchronous multi-copy technique allows for copying data from the host to several devices and vice versa simultaneously, GPU-direct introduces the ability to transfer data between the memory of distinct GPUs without even involving the CPU. It also features the ability

to access GPU memory located in a different device inside a kernel call.

Despite the fact that more implementations can be realized, we want to limit our analysis to the following possibilities:

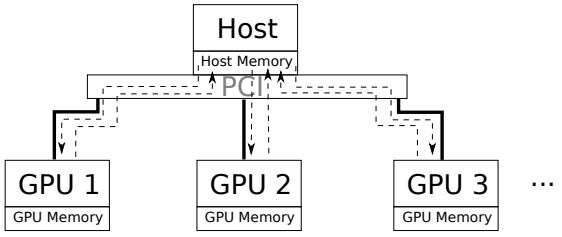
1. **Asynchronous Multicopy (AMC)** Like in the classical approach, the host memory serves as communication facility. The performance is improved by allowing for asynchronous data transfers from and to the distinct graphics processing units. The data is put in streams, that are then handled independently and without synchronization. This allows the simultaneous data transfer between host and multiple devices. While for computing the next iteration requires the communication of all components of the iteration vector, every GPU sends only the new values of the respectively updated components back to the host. We want to stress that this approach is only possible due to the asynchronism in the iteration method, otherwise the performance would suffer from synchronization barriers managing the update order. This approach, illustrated in Figure 4.9a may especially be interesting for architectures where different GPUs are accessed with different bandwidths, or the distinct GPUs processing with different speed.
2. **GPU-Direct Memory Transfer (DC)** One feature of CUDA 4.0 is the GPU-direct, allowing the data transfer between different GPUs via PCI without involving the host. If the complete iteration vector is stored on one GPU, say the master-GPU, it can be transferred to the other devices without using the main memory of the host. Again, after every GPU has completed the kernels, the updated components are copied back to the memory of the master-GPU, serving as central storage, see Figure 4.9b. While connection to the master-GPU now becomes the bottleneck, the CPU of the host system is not needed, and can be used for other tasks.
3. **GPU-Direct Memory Kernel Access (DK)** The possibility to access data located in another device inside a kernel allows even another implementation, avoiding any explicit data transfer in the code. In this algorithm, the components of the iteration vector are exclusively stored in the memory of the master-GPU, and the kernels handled by the different devices directly access the data in the master-GPU, see Figure 4.9c. Like in the GPU-direct memory transfer implementation, the CPU of the host system is not needed, and can be used for other tasks.

4.7. Experiments of Block-Asynchronous Iteration on Multi-GPU Systems

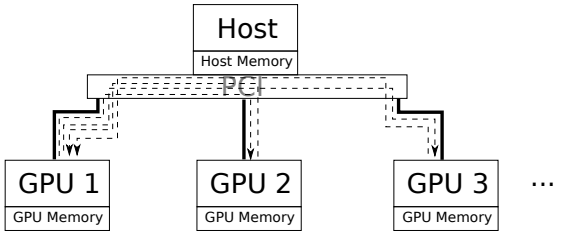
In experiments we now want to analyze the properties of the different approaches to block-asynchronous iteration using multi-GPU systems. First, we focus on the impact of using multiple GPUs on the convergence rate and the performance. Therefore, we apply the different approaches to the test matrix Trefethen_20000 matrix (see Appendix B), which is due to its size and structure suitable for the experiment.

In the first test, we use the asynchronous multicopy (AMC, see Section 4.6) with one, two, three and four GPUs, which can efficiently be realized since the used Supermicro system features multiple PCI controllers, see Appendix C.1.

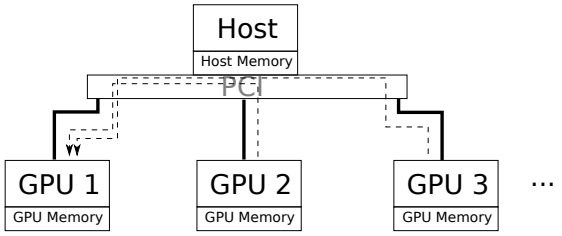
In Figure 4.10a we report the iteration times, in Figure 4.10b the respective residual improvements. We observe that the initialization process takes about 0.8 seconds, independent of the number of devices. Since it includes the memory transfer of the system matrix to the respective devices, using multiple GPUs and copying only the respectively



(a) Asynchronous multicopy (**AMC**)

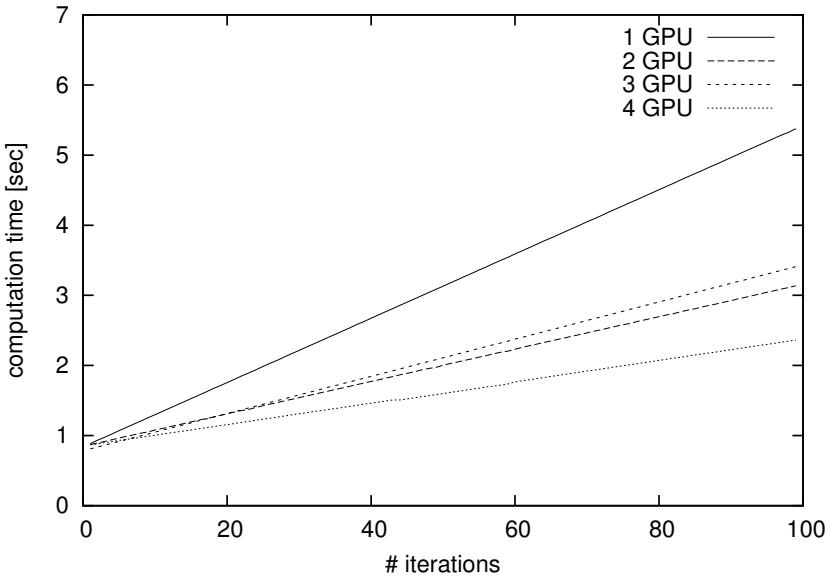


(b) GPU-direct memory transfer (**DC**), GPU1 serves as master-GPU.

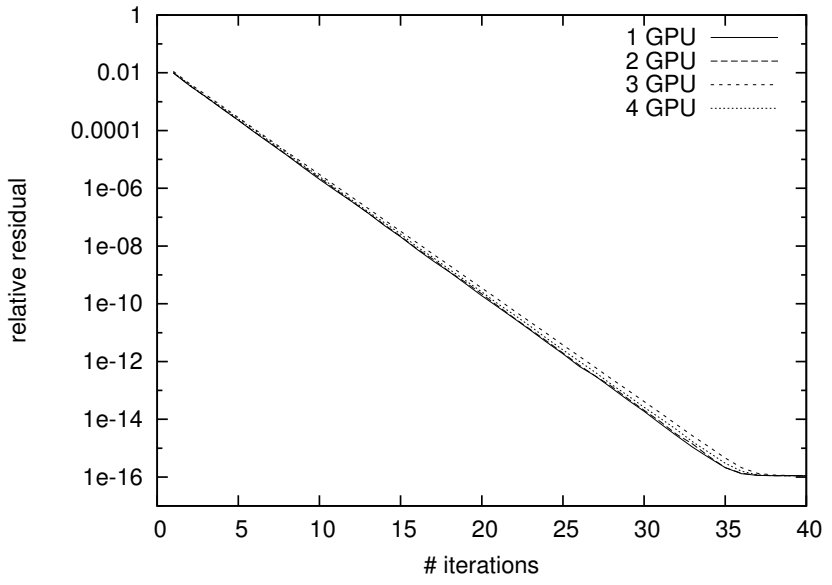


(c) GPU-direct memory kernel access (**DK**), GPU1 serves as master-GPU.

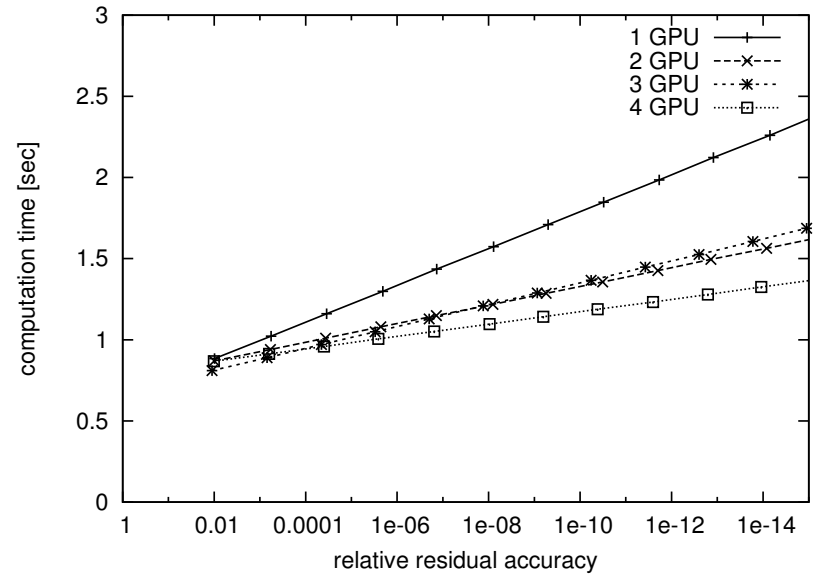
Figure 4.9.: Visualizing the different Multi-GPU memory handling. The dotted lines indicate the explicit data transfers.



(a)



(b)



(c)

Figure 4.10.: Performance of asynchronous multicopy (AMC) using different device numbers.

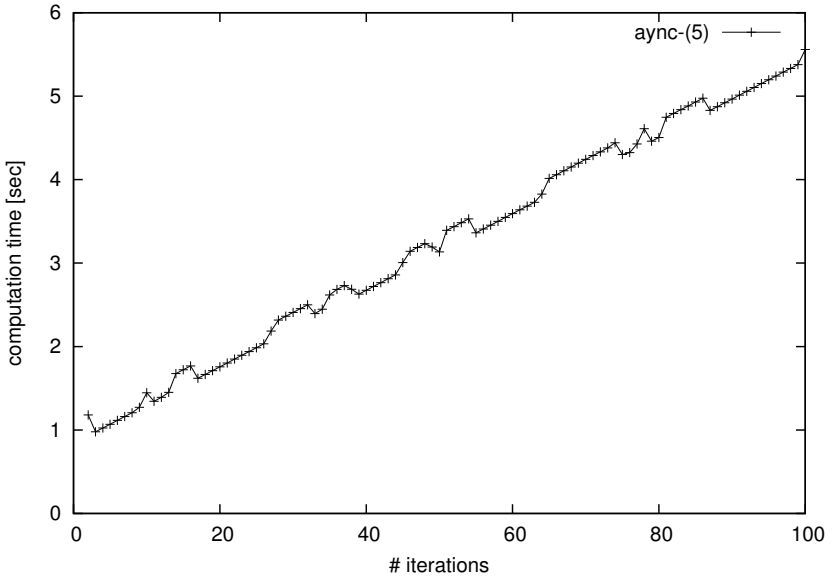


Figure 4.11.: Performance differences depending on data locality using one GPU for the asynchronous multicopy.

necessary matrix parts to the different GPUs in parallel via different IO interfaces may trigger a small reduction in the initialization process. Furthermore, we observe that the average time needed for one global iteration can be decreased considerably when switching from one to two GPUs: Drawing the initialization overhead, even the optimal speedup factor of two can be achieved. This stems from the fact, that each GPU only handles half the components, and due to the asynchronous data transfer using different PCI controller and different PCI connections, the two devices can add their performance.

Switching from two to three GPUs we observe a performance decrease. While we iterate still faster than when using only one GPU, we are about 20 % slower than for the dual-GPU case. To understand a possible explanation for this effect, we have to consider the used hardware. The Supermicro system consists of two CPUs connected via QPI, where each CPU is connected to respectively two GPUs (see Appendix C.1). This architecture implies that, depending on the location of the data in main memory, when computing on a GPU, the data has to be transferred via the QPI connection between the two CPUs. Eventually, the data has to be transferred even twice. This case occurs if the data is located in the Memory of CPU1, the algorithm uses CPU2 for the host operations, but the used GPU is connected to CPU1. Then, the data has to be accessed via the QPI connection, and then communicated to the GPU via the QPI again.

For a better understanding of this issue, we conduct an experiment using the async-(5) algorithm on GPU0 connected to CPU1 We then execute the algorithm several times for different iteration counts and report the overall execution time in Figure 4.11. Beside the number of iterations, the only parameter we change is the CPU handling the host operations and the location of the data in main memory. Either the memory of CPU1 or CPU2 is chosen, which leads to different communication effort. In the results we can identify two distinct lines where the timing results are located on. This suggests the speculation that depending on the location of the data in main memory, the overall execution time is increased by the data transfers.

If we now consider the asynchronous multiycopy algorithm using three GPUs, we necessarily include the communication via the QPI. We may consider this as an explanation

why the performance decreases when switching from two to three GPUs in Figure 4.10a.

Increasing the GPU number from three to four, we again benefit from the reduced component number handled per device. This stems from the fact that the application now includes the communication via the QPI anyway, and using more GPUs pays off, as the number of components handled by one GPU and the therewith associated average time for one global iteration is decreased. Although this configuration outperforms the dual-GPU performance, the speedup is, due to the additional communication overhead, considerably smaller than the factor of two.

For the convergence characteristics, using less devices may be beneficial. A possible reason is that additional hardware components may, despite the faster component handling, trigger additional delay in the communication, having negative impact on the convergence rate. Nevertheless, we observe in Figure 4.10b that the variations are negligible, and an interesting observation is, that using four instead of three GPUs even increases the convergence rate. The reason may again be due to the system architecture, where always two GPUs are connected to one CPU. In Figure 4.10c we combine the results for iteration timing and convergence by reporting the average computation time needed to obtain a certain approximation accuracy. We can conclude, that the slightly slower convergence rate with respect to iteration numbers has almost no impact: the time-to-solution performance is very similar to the iteration performance.

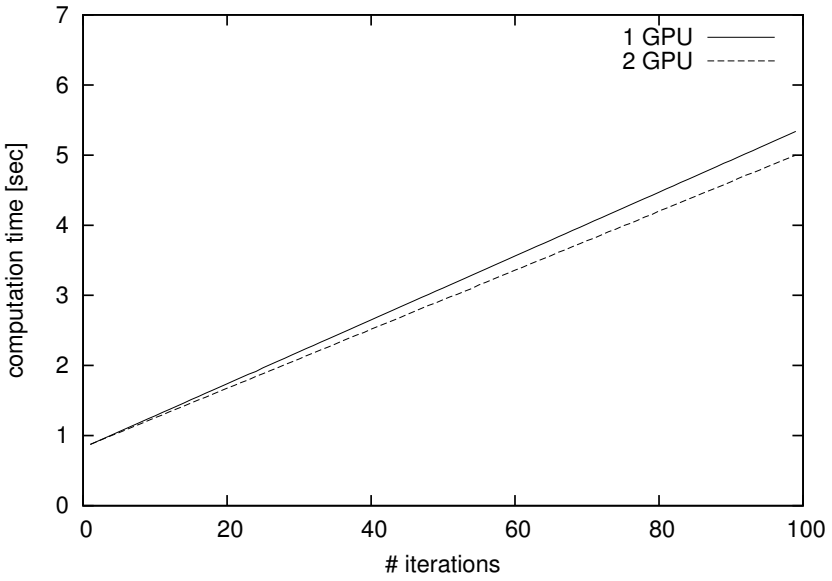
We now want to analyze the other implementations leveraging the GPU-direct technology. Unfortunately, for the specific architecture of the Supermicro system (see Appendix C.1) CUDA does not yet support GPU-direct between more than two GPUs as CUDA's GPU-GPU communication is only supported for GPUs connected to the same CPU [NVI09]. Therefore, we have to limit our analysis of the GPU-direct memory transfer (DC, see Section 4.6) and the GPU-direct memory kernel access (DK, see Section 4.6) to the cases where we are using one or two GPUs.

First, we consider the GPU-direct memory transfer (DC). In Figure 4.12 we report the iteration timings, the residual improvement and the time-to-solution.

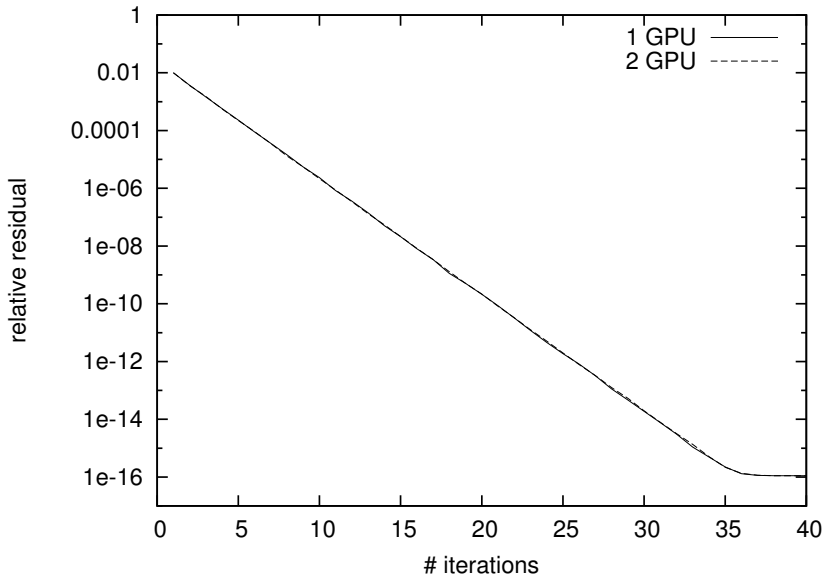
We observe, that the difference in any of the metrics is negligible. The average iteration timing is improved by less then 10 %, the convergence behavior shows almost no difference, and also for the time-to-solution we notice only small improvement. Considering the implementational overhead and the additional power usage when computing on multiple GPUs we may conclude that this approach is not beneficial for the given experiment setup.

The GPU-direct memory kernel access (DK) differs from the GPU-direct memory transfer approach in the way that the remote GPU cores only fetch the respectively needed data, and write back. The memory copy transferring the complete set of the remotely handled components in the DC approach is replaced by very particular communication. In Figure 4.13a we report the average iteration times and observe that the results are very similar to the DC approach. This was expected due to the very similar algorithm design. A difference can be observed for the convergence rate (see Figure 4.13b): the component-specific communication may trigger additional communication delay, resulting in slower convergence. Like in the DC approach, the term average iteration number in this case really is an average including large variations, since the components handled locally may be updated far more often than the components handled by the remote GPU. The result is a slower convergence with respect to average iteration numbers. This also impacts the time-to-solution performance: the improvement by using an additional device is almost outweighed by the slower convergence (Figure 4.13c).

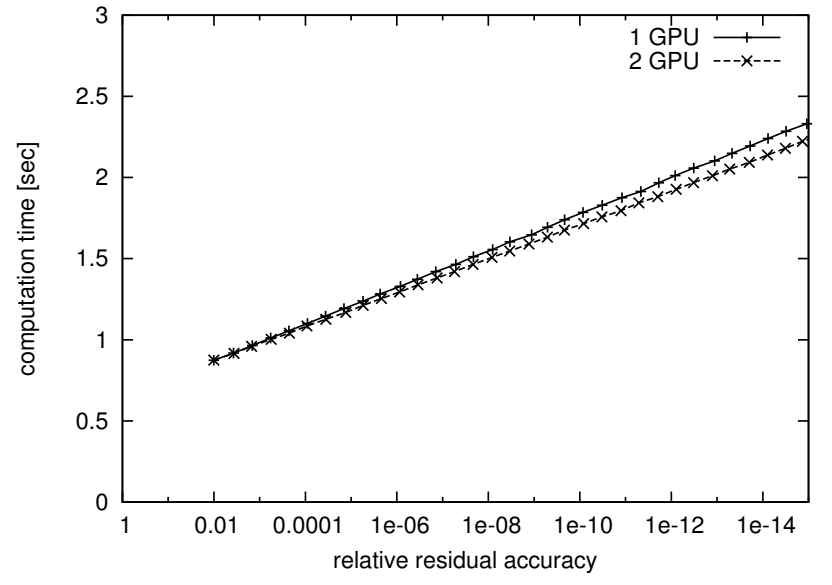
Since we have seen that for all implementations, the solution approximation accuracy almost linearly depends on the run-time, for comparing the performance of the different



(a)

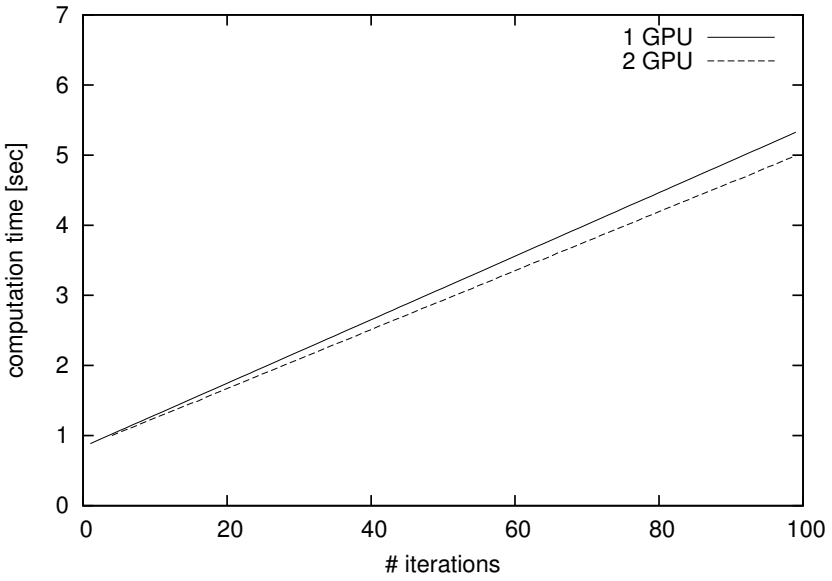


(b)

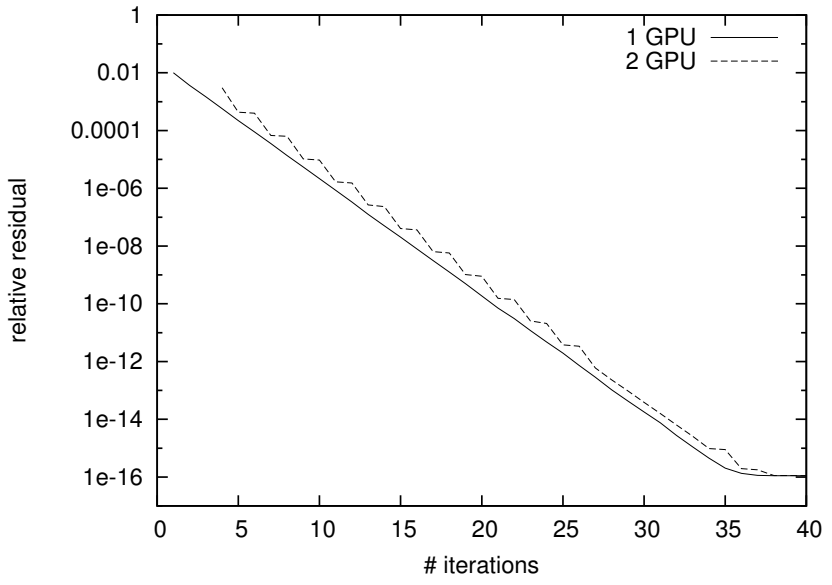


(c)

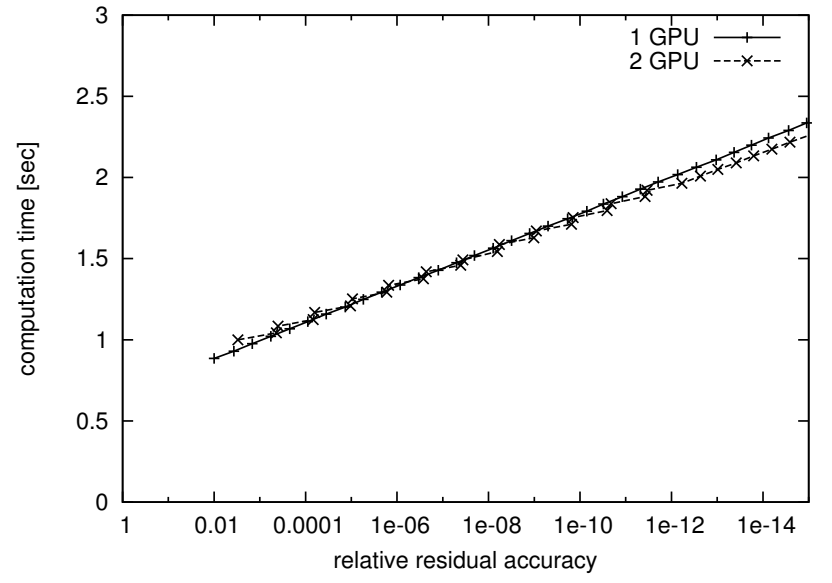
Figure 4.12.: Performance of GPU-direct memory transfer (DC) using different numbers of devices.



(a)



(b)



(c)

Figure 4.13.: Performance of GPU-direct memory kernel access (DK) using different numbers of devices.

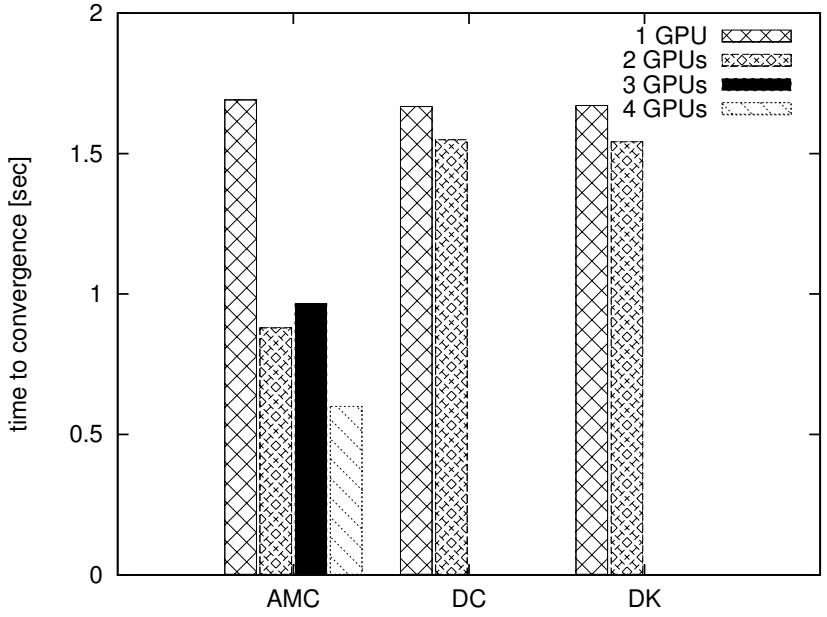


Figure 4.14.: Time-to-solution for the different Multi-GPU implementations for test matrix Trefethen_20000.

implementations it is sufficient to report the time-to-convergence. In Figure 4.14 we subtracted the respective initialization overhead. For the case using only one GPU, the DC and DK approach is slightly faster than the asynchronous multicopy since the iteration vector resides in the GPU memory and is not transferred back and forth between CPU and GPU. Other than that, the algorithms of asynchronous multicopy, GPU-direct memory transfer and GPU-direct memory kernel access do not differ when running on only one GPU. As soon as we include a second GPU, the asynchronous multicopy performs considerably faster. The total run-time is almost cut in half, while for the GPU-direct based implementations, only small improvements can be observed. The reason for the excellent speedup of the AMC becomes possible by using different PCI connections, such that every GPU may leverage the complete bandwidth, which is essential since the application seems to be memory bound. Using the memory located in one of the GPUs, like we do in the GPU-direct approaches, puts a lot of pressure on the PCI connection of this GPU: all data has to be transferred via this connection.

4.8. Block-Asynchronous Iteration for Sparse Systems

In the following sections, we want to introduce a matrix storage format that allows a more efficient usage of block-asynchronous iteration when targeting sparse systems. The motivation is that for systems where most matrix entries equal zero, using a dense representation induces considerable overhead to the computational complexity and the required memory. The latter one may be crucial for the communication when using GPU-accelerated systems. While the widely used CRS format is usually very efficient when handling sparse systems, we apply small modifications to this format to optimize it for the block-asynchronous algorithm.

4.8.1. CRS Format

When using finite elements for the discretization of partial differential equations, most of the coefficients in the obtained linear system equal zero. Therefore, it is usually not necessary to store the complete matrix. While storing only the nonzero entries usually leads to

significant performance improvement when conducting computations, memory constraints often do not allow any other choice, e.g. if the system has to be stored in limited device memory.
The challenge when storing only the nonzero entries is how to remain the locality information. While the most intuitive idea of adding two arrays containing the row- and column-index of every nonzero entry would lead to a memory demand of three arrays of the length nnz where nnz is the number of nonzero elements in the matrix, a more efficient and widespread used format is the "compressed row storage" (CRS). The following description is largely taken from [BBC+94].

The underlying idea of the compressed row storage is to put the subsequent nonzeros of the matrix rows in contiguous memory locations. Assuming a nonsymmetric sparse matrix A, it creates three vectors: one for floating-point numbers val (of length nnz), and the other two for integers where col (of length nnz) contains the column indices and row (of length $n+1$) pointers to the first element in every row. The val vector stores the values of the nonzero elements of the matrix A, as they are traversed in a row-wise fashion. The col vector stores the column indexes of the elements in the val vector. That is, if $val(k) = a_{i,j}$ then $col(k) = j$. The row pointer vector stores the locations in the val vector that start a row, that is, if $val(k) = a_{i,j}$ then $row\ (i) \leq k < row(i+1)$. By convention, $row(n+1) = nnz$, where nnz is the number of nonzeros in the matrix A. The storage savings for this approach may be significant: Instead of storing n^2 elements, memory for $2nnz + n + 1$ elements is needed. While for dense systems ($nnz = n^2$) this approach even causes an overhead, it is beneficial for sparse systems where where typically $2nnz << n^2$.

But despite the efficient storage and the good performance for most algebraic operations, the CRS format has one drawback that may for some algorithms, especially relaxation schemes, be crucial: There exists no direct access to the diagonal elements. The reason is, that there is no information stored about whether an element is on the diagonal. To get the diagonal element of a row, it is necessary to pass through the part of the col array where the column information of the respective row can be found until the diagonal element is identified. This not only demands for if statements in the code that are difficult to parallelize, but also for a high number of memory jumps that usually cause poor performance.

In order to avoid this problem we introduce a matrix format that is optimized for diagonal-focused algorithms like the Jacobi method (see Section 2.4.1). Therefore we split the matrix into two parts, one containing the diagonal block elements with adjustable block-size, and one containing the remaining off-diagonal part. In the matrix containing the diagonal blocks, we reorder every row such that the diagonal element is always first. It should be mentioned though, that this reordering is only beneficial for our application where fast access to the diagonal elements is crucial. For other operations, it may cause poor performance due to additional memory jumps.

4.8.2. Modified Sparse Matrix Storage for Asynchronous Iteration Method

Algorithm 16 Asynchronous iteration for sparse matrices stored in CRS format.

```
1: for all (index ∈ {1...n}) do
2:   int start = row[index];
3:   int end = row[index + 1];
4:   double tmp = 0.0;
5:   double diag = 0.0;
6:   for (i = start; i < end; i++) do
7:     if (col[i] == index) then
8:       diag = val[i];
9:       tmp+ = val[i] · x[col[i]];
10:    else
11:      tmp+ = val[i] · x[col[i]];
12:    end if
13:  end for
14:  x[index] = x[index] + 1.0/diag · (b[index] − tmp);
15:  {/*————————— possibility to add more local iterations ————————— */}
16:  for (iters = 0; iters < local_iters; iters++) do
17:    double tmp = 0.0;
18:    double diag = 0.0;
19:    for (i = start; i < end; i++) do
20:      if (col[i] == index) then
21:        diag = val[i];
22:        tmp+ = val[i] · x[col[i]];
23:      else
24:        tmp+ = val[i] · x[col[i]];
25:      end if
26:    end for
27:    x[index] = x[index] + 1.0/diag · (b[index] − tmp);
28:  end for
29: end for
```

In Algorithm 16, a straight-forward parallel implementation of the asynchronous iteration for sparse matrices stored in CRS format can be found. But especially when targeting highly parallelized devices like GPUs, the *if-statements* identifying the diagonal element may cause poor performance. Since the diagonal element is located at a different place for every row, also the SIMT model provided by CUDA may not improve the situation, see Section 4.1. A possible expedient to this issue is to replace the requests by a feature of the C programming language, see Algorithm 17. However, when executing the code on GPUs, we observe almost no performance difference between the codes, which suggests that the additional multiplication and type conversion generates an overhead comparable to the version using predication.

The situation can finally be improved by reordering the elements in one row, such that the diagonal element is always first. Additionally, performance may benefit from splitting the matrix into parts such that diagonal blocks can be used for local iterations like in Section 4.2.
Therefore, we propose the following matrix layout for asynchronous iteration methods:

1. The matrix currently stored in CRS format is split into two matrices, that are then again stored in CRS format.
2. One matrix contains only diagonal blocks, the other one the off-diagonal blocks.

Algorithm 17 Asynchronous iteration for sparse matrices stored in CRS format, the layout of the *if-conditions* is replaced by the expression in lines 7 and 16.

```
1:  for all (index ∈ {1...n}) do
2:    int start = row[index];
3:    int end = row[index + 1];
4:    double tmp = 0.0;
5:    double diag = 0.0;
6:    for (i = start; i < end; i++) do
7:      diag+ = int(col[i] == index) · val[i];
8:      tmp+ = val[i] · x[col[i]];
9:    end for
10:   x[index] = x[index] + 1.0/diag · (b[index] − tmp);
11:   {/*————————— possibility to add more local iterations —————————— */}
12:   for (iters = 0; iters < local_iters; iters++) do
13:     double tmp = 0.0;
14:     double diag = 0.0;
15:     for (i = start; i < end; i++) do
16:       diag+ = int(col[i] == index) · val[i];
17:       tmp+ = val[i] · x[col[i]];
18:     end for
19:     x[index] = x[index] + 1.0/diag · (b[index] − tmp);
20:   end for
21: end for
```

The block size is chosen according to the matrix decomposition that is optimized to hardware and problem-dependent parameters.

3. If one of the matrices contains no elements in one row, a zero-element is artificially created in this row. This allows for processing the matrices uniformly in the block-asynchronous iteration algorithm, and causes negligible memory and computational overhead.

4. The matrix containing the diagonal blocks is reordered in memory, such that the diagonal element of every row is always first. This allows easy access to the diagonal elements, since now every *row* element points to the diagonal element in the respective row. The order of the remaining elements in the row is immaterial, and can be left unchanged.

5. The memory overhead triggered by the new data layout is at most $2n + 1$ additional integers and $2n$ floating point numbers, which is the case if one of the created matrices is the zero-matrix. Usually, for both matrices there exist elements in all rows, which results in a memory overhead of only one additional integer, since now the nonzero information of two matrices has to be stored.

Visualized in Figure 4.15, the introduced data layout has the potential to significant performance increase since the original code can be redesigned, see Algorithm 18. We may denote this modified CRS format as *Block-CRS*. Note that it always has to be adapted to the block decomposition of the matrix. An open question in this context is the computational cost of rearranging the data. Depending on the specific matrix properties, the overhead caused may have considerable impact on the overall algorithm performance.

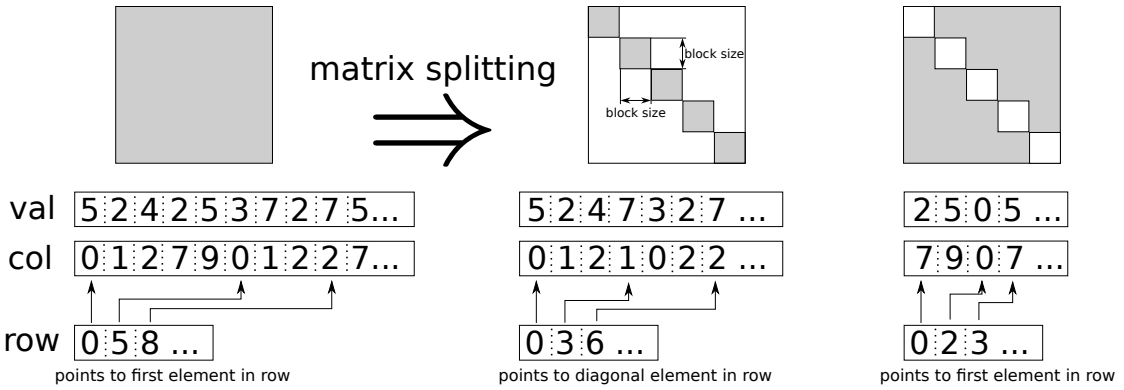
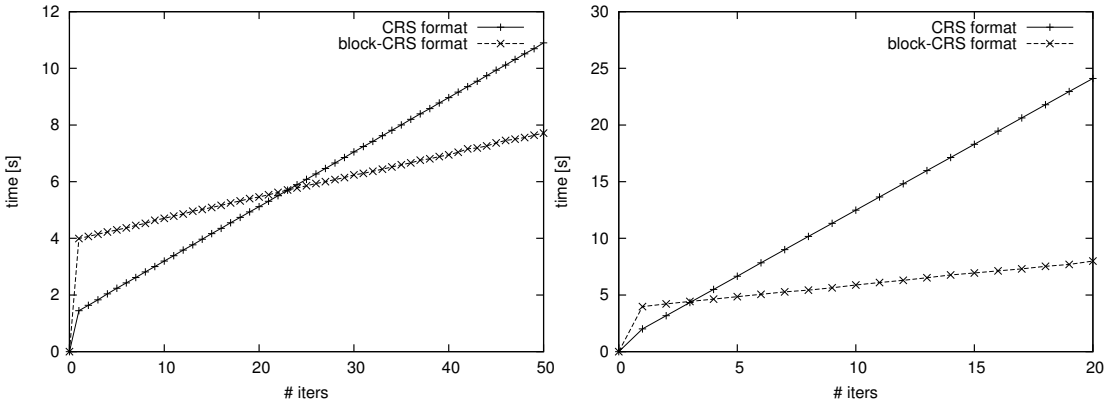


Figure 4.15.: Splitting a CRS matrix into a diagonal-block and a non-diagonal-block part for a block-size of 3 elements. Note that in the matrix containing the diagonal blocks, the *row* vector element points to the diagonal element in every row, and the non-diagonal-block matrix contains an artificially created zero element in the second row.

Algorithm 18 Asynchronous iteration for the block-CRS format. Notice that ind_diag is the index where the diagonal block starts, local_iter is the number of the local iterations, and _diag, _offdiag denote the matrices containing the diagonal blocks and the offdiagonal blocks, respectively.

Convert the CRS data layout into the block-CRS data layout;
for all $(index \in \{1 \dots n\})$ **do**
  initialize ind_diag with start of the block $index$ is located in;
  int $start = row\_offdiag[index]$;
  int $end = row\_offdiag[index + 1]$;
  double $tmp = 0.0$;
  double $v = 0.0$;
  double $diag = 0.0$;
  for $(i = start; i < end; i++)$ **do**
    $v+ = val\_offdiag[i] \cdot x[col\_offdiag[i]]$;
  end for
  $start = row\_diag[index]$;
  $end = row\_diag[index + 1]$;
  for $(i = start; i < end; i++)$ **do**
    $tmp+ = val\_diag[i] \cdot x[col\_diag[i]]$;
  end for
  $v = b[index] - v$;
  {/*———————— possibility to add more local iterations ———————— */}
  $local\_x[index - ind\_diag] = x[index] + (v - tmp)/val\_diag[start]$;
  for $(j = 0; j < local\_iters; j++)$ **do**
    $tmp = 0.0$;
    for $(i = start; i < end; i++)$ **do**
      $tmp+ = val\_diag[i]\ local\_x[col\_diag[i] - ind\_diag]$;
    end for
    $local\_x[index - ind\_diag]+ = (v - tmp)/val\_diag[start]$;
  end for
  $x[index] = local\_x[index - ind\_diag]$;
end for



(a) async-(1) (b) async-(5)

Figure 4.16.: Block-asynchronous iteration performance for test matrix A318 using CRS and block-CRS data layout, respectively.

4.9. Experiments on Block-Asynchronous Iteration for Sparse Systems

In this section we want to compare the performance of the two block-asynchronous iterations for sparse systems using the CRS and the block-CRS format introduced in the last section. Using the block-CRS format in Algorithm 16 triggers, additionally to the GPU initialization, some more overhead at the beginning since the linear system matrix has to be converted into the new format. While this overhead is small for our large but very sparse test case, depending on the matrix structure it may become a crucial factor in the algorithm's performance. Once this is arranged, we may benefit from the new format optimized to the method's design. This may be true for the plain asynchronous iteration as well as the block-asynchronous iteration. In order to analyze this trade-off, we conduct experiments on the large sparse system A318 (see Appendix B). Neglecting small rounding effects that may occur, we know that the same global iteration count generates the same solution approximations. In Figure 4.16 we report the computation time for different numbers of global iterations.

Analyzing the runtimes we realize that the overhead by the data conversion into the block-CRS format may pay off for the block-asynchronous iteration using multiple local iterations as well as the block-asynchronous iteration using only one local update. For the latter one (see (async-(1), Figure 4.16a), using the block-CRS data layout is beneficial when conducting more than 25 iterations. Comparing the data given in Table 4.8 we realize that the improvements using the modified format results for high iteration numbers in speedups of more than two. As expected, the block-asynchronous using 5 local iterations (async-(5)) benefits even more from modifying the data layout (Figure 4.16b). We observe that the overhead of the conversion approximates one iteration of the CRS based implementation. This causes overhead of 100% for the first iteration using async-(5), but already when performing three iterations, this overhead is compensated for. The performance improvement per iteration for async-(5) approximates 80 %, which results in substantial speedups for high iteration numbers (see Table 4.8).

4.10. Problem-Aware Block-Asynchronous Iteration

In the previous sections of this chapter we have shown that using block-asynchronous iteration can be beneficial when working on heterogeneous highly parallel hardware. We now want to investigate whether adapting the method to a specific problem is possible,

method	CRS [sec]	block-CRS [sec]	reduction	overhead CRS	overhead block-CRS
async-(1)	0.19453	0.07063	64 %	1.2114	3.8482
async-(5)	0.90580	0.21217	82 %	1.1260	3.7623

Table 4.8.: Timings per iteration for CRS and block-CRS based asynchronous iterations. The results are average timings for one global iteration.

and whether this pays off in terms of convergence behavior and runtime performance. Since many linear systems that have to be solved in scientific computing arise from the discretization of partial differential equations, we consider two experiment setups, where the block-asynchronous iteration is either adapted to a given system of linear equations by using the matrix properties, or by interacting with the discretization, which is possible if the discretization is flexible, and the underlying partial differential equation itself is given.

The respective underlying ideas are:

- Assuming a given system of linear equations with a specific matrix structure, it may be beneficial to adapt the block-asynchronous iteration to this structure by either using weights for the local iterations on the subdomains, or by controlling the number of local iterations according to the diagonal dominance of the distinct blocks. Similar to weighting in hybrid parallel smoothers [HY00], the fundamental idea of using weights in block-asynchronous iteration is to control the influence of the local iterations on the subdomains by modifying the iteration matrix. Adapting the number of local iterations in contrast applies no modifications to the iteration matrix, but accounts for the matrix structure by conducting different iteration counts on the distinct subdomains.
- Targeting the solution of a partial differential equation where the discretization process is known and allows for some adjustments, it may be beneficial to account for the discretization parameters in the block-asynchronous method. Particularly, in case of using a uniform finite element discretization, e.g. finite differences, the block-size can efficiently be adapted to the element number per direction to reduce the off-block parts in the iteration matrix. But also the discretization may be adapted to the iteration method and the used hardware.

In the following sections, we will first motivate the use of weights by the successful application in multigrid methods. We then introduce weighting techniques for the block-asynchronous iteration, show that the convergence is preserved (Section 4.10.2.2 and 4.10.2.4) and then analyze the performance increase in numerical experiments. In Section 4.10.4 we proceed with introducing an iteration scheme where the number of local iterations on the distinct blocks is chosen with respect to the ratio between block and off-block parts. In Section 4.10.5 we investigate in experiments how adapting the number of local iterations pays off.

We then target the problem of adapting the block-asynchronous iteration to a given partial differential equation. After introducing some fundamental concepts in Section 4.10.6 we show in Section 4.10.8 for an academic problem that not only adapting the iteration to the discretization pays off, but we may also benefit from discretizing with respect to the hardware and implementation.

4.10.1. Weights in Multigrid Smoothers

Using weights in iterative relaxation schemes is a well known and often applied technique to improve the convergence. An example is the classical successive over-relaxation

(SOR [Saa03], see Section 2.4.3), which may improve the convergence rate of the underlying Gauss-Seidel algorithm by using weights. Other prominent examples include block-smoothers in multigrid methods for finite element discretizations [BFKMY11]. In this case, the parallelized Block-Jacobi- or Block-Gauss-Seidel smoothers are weighted according to the block decomposition of the system of linear equations.

While Chazan and Miranker have introduced a weighted asynchronous iteration similar to SOR [CM69], it becomes of interest as whether the block-asynchronous iteration benefits from weighting methods similar to those applied to block smoothers. The motivation is that the off-block matrix entries are neglected in the local iterations performed on the subdomains. To account for this issue it may be beneficial to weight the local iterations. This can be achieved either by using ℓ_1-weights, by a technique similar to the ω-weighting proposed by Chazan and Miranker in [CM69] but adapted to the block design, or by a combination of both. For this purpose we will first derive different methods, analyze the convergence properties, and then report experimental results.

4.10.2. Weighting in Block-Asynchronous Iteration

The underlying block-asynchronous iteration which we consider is the method we introduced in Section 4.2. Due to the design of the algorithm based on a block decomposition of the iteration matrix, it may be beneficial to apply weighting techniques similar to those applied in block-smoothers for multigrid methods [BFKMY11]. We may also benefit from employing an ω-weighting technique for the block-asynchronous iteration like the one used in block-SOR. The common idea of all approaches is to control the influence of the local iterations. Especially when targeting strongly coupled systems, where there exist significant matrix entries away from the diagonal, the local iterations may trigger poor convergence, since they only account for the matrix entries in the respective block.

To examine the topic of weights in block-asynchronous iteration we first introduce some notation, that may simplify the further analysis. Splitting the matrix A into blocks, we use $A_{l,l}$ for the matrix block located in the l-th block row and the l-th block column. Furthermore, we introduce the index sets Ω_l where

$$\Omega = \bigcup_{l=1}^{p} \Omega_l = \{1, 2 \dots n\},$$

and $\Omega_i \cap \Omega_j = \emptyset \; \forall i \neq j$ consistent to the block decomposition of the matrix A. Using this notation, Ω_l contains all indices j with $T_S(l) \leq j \leq T_E(l)$ where $T_S(l)$ (respectively $T_E(l)$) denotes the first (last) row and column index of the diagonal block $A_{l,l}$. We now define the sets

$$\Omega^{(i)} = \{j \in \Omega_l : i \in \Omega_l\},$$
$$\Omega_0^{(i)} = \{j \notin \Omega_l : i \in \Omega_l\}.$$

Hence, for block $A_{l,l}$, $\Omega^{(i)}$ contains the indices with corresponding columns being part of the diagonal block of row i while $\Omega_0^{(i)}$ contains the indices of the columns that have no entries in the block. This way, we can decompose the sum of the elements of the i-th row:

$$\sum_j a_{i,j} = \underbrace{\sum_{j=1}^{k_S-1} a_{i,j}}_{\text{off-block columns}} + \underbrace{\sum_{j=T_S}^{k_E} a_{i,j}}_{\text{block columns}} + \underbrace{\sum_{j=T_E+1}^{n} a_{i,j}}_{\text{off-block columns}} \tag{4.2}$$

$$= \underbrace{\sum_{j\in\Omega^{(i)}} a_{i,j}}_{\text{block columns}} + \underbrace{\sum_{j\in\Omega_0^{(i)}} a_{i,j}}_{\text{off-block columns}} . \tag{4.3}$$

We may now define an indicator θ such that for all rows i

$$|a_{i,i}| \geq \theta \sum_{j\in\Omega_0^{(i)}} |a_{i,j}| \tag{4.4}$$

indicates a certain quality of the parallel partitioning of the matrix A. Large values for θ imply that most of the relatively significant entries in every row are on the diagonal [BFKMY11]. For the block-asynchronous iteration using the matrix block decomposition this means that the off-block entries, which are neglected in the local iterations, are relatively small.

4.10.2.1. ω-weighting for Block-Asynchronous Iteration

Similar to the ω-weighted asynchronous iteration (see Theorem 3.2.5 Section 3.2, [CM69]), it is possible to use ω-weights for the block structure in the block-asynchronous approach. In this case, the solution approximation of the local iterations is weighted when updating the global iteration values. The parallel algorithm for the component updates in one matrix block is outlined in Algorithm 19.

Algorithm 19 Basic principle of using ω weights in block-asynchronous iteration.

1: update component i:
2: $s := d_i + \sum_{j\in\Omega_0^{(i)}} b_{i,j}x_j$ {off-diagonal part}
3: $x^{local} = x$
4: **for** $(k = 0;\ k < local\_iters;\ k++)$ **do**
5: $\quad x_i^{local} := s + \sum_{j\in\Omega^{(i)}} b_{i,j}x_j^{local}$ {using the local updates in the block}
6: **end for**
7: $x_i = \omega x_i^{local} + (1-\omega)x_i$

For this algorithm it is difficult to derive one explicit iteration matrix, as for the local iterations $B = I - D^{-1}A$ is applied, and for the global updates $B_\omega = I - \omega D^{-1}A$ is used.

4.10.2.2. Convergence of ω-weighted Block-Asynchronous Iteration

Theorem 4.10.1. *Suppose the asynchronous iteration fulfilling (a) and (b) of Condition 3.2.1 is modified by using weights in the block-asynchronous iteration like in Algorithm 19 where local_iters denotes the number of local iterations on every subdomain.*
If the spectral radius of $\rho(|B|) = \alpha < 1$, then the ω-weighted block-asynchronous iteration converges for all ω with $0 < \omega < \frac{2}{\alpha+1}$.

Proof. If the condition $\rho(|B|) = \alpha < 1$ is fulfilled, we know from Theorem 3.2.3 that the respective iteration scheme is convergent to the unique solution x^*. Furthermore, from Theorem 3.2.5 we get for $0 < \omega < \frac{2}{\alpha+1}$ the existence of $\beta < 1$ with $|B_\omega|v = \beta v$, and

consequently the ω-weighted iteration also converges to the unique solution x^* [CM69]. Hence, we have two different iteration schemes with the iteration matrices B respectively B_ω that are convergent for any update pattern. The method proposed in Algorithm 19 is an iteration scheme where these two iteration methods are combined such that for every component update either B or B_ω is applied. In fact, the ω-weighted block-asynchronous iteration arises as a very specific case of component update pattern and specific sequence of applying B and B_ω.

To analyze the convergence of this method we show that in case of two converging asynchronous iteration schemes based on the original and the weighted iteration matrix, also any combination of these two schemes using an arbitrary pattern of component updates and choice of iteration matrix converges.

Let $e^k = x^* - x^k$ be the error in the t-th iteration. Like in the proof of Theorem 3.2.2 we consider the first $\bar{s}$ iterates in the process, where $\bar{s}$ is the upper bound for the shift function. We have $\rho(|B|) = \alpha < 1$ and $0 < \omega < \frac{2}{\alpha+1}$, and now define $\xi := \max\{\alpha, \beta\}$ for the $\beta = |1-\omega| + \alpha\omega < 1$ from Theorem 3.2.5. Obviously, $\xi < 1$. From Theorem 3.2.3 and Theorem 3.2.5 we can deduce that there exists a positive vector $v > 0$ with $|B|\, v \leq \xi v$ and $|B_\omega|\, v \leq \xi v$. Since all components of v are positive, there exists a positive value γ such that $|e^k| \leq \gamma v$ for $0 \leq t \leq \bar{s}$ (component-wise, $|e_j^k| \leq \gamma v_j \ \forall j$). We now consider the update of component i using B or B_ω and any of these $\bar{s}$ vectors forming the first $\bar{s}$ iterates. Let

$$\tilde{B} = \begin{cases} B & \text{if the iteration matrix } B \text{ is applied} \\ B_\omega & \text{if the iteration matrix } B_\omega \text{ is applied} \end{cases}$$

We may assume without loss of generality that $b = 0$ in (2.1) so that $x^* = 0$ and $\tilde{B}x_i^k = -\tilde{B}e_i^k$. Then, the update satisfies

$$\begin{aligned} |e_i^{k+1}| = |x_i^* - x_i^{k+1}| &= |0 - x_i^{k+1}| \\ &= |\sum_{j=1}^{n} \tilde{b}_{i,j} x_j^{k-s(t,j)}| \leq \sum_{j=1}^{n} |\tilde{b}_{i,j} e_j^{k-s(t,j)}| \\ &\leq \sum_{j=1}^{n} |\tilde{b}_{i,j}||e_j^{k-s(t,j)}| \leq \xi\gamma v_i. \end{aligned}$$

If t_1 is the first instance after $\bar{s}$ for which all components have been updated using either the iteration matrix B or B_ω, then $|e^{k_1}| \leq \xi\gamma v$. Moreover, $|e^k| \leq \xi\gamma v$ for all $t \geq t_1$. Similarly, if t_2 is the next instance after t_1 for which all components have been updated a second time using one of the iteration matrices, then $|e^k| \leq \xi^2\gamma v$ for all $t \geq t_2$. This way we obtain that the error $|e^k| = |x^* - x^k|$ converges to zero for $t \to \infty$. Hence, for any update pattern and any sequence of iteration matrices the solution approximation is convergent to x^*, the unique solution of (2.1).

□

4.10.2.3. ℓ_1-weighting in Block-Asynchronous Iteration

Using the notation for the local and the global parts of the matrix (see (4.2)), we can introduce a weighting technique, that is usually referred to as ℓ_1-weighting [BFKMY11] (see Figure 4.17). Classically applied to Block-Jacobi and Gauss-Seidel relaxation methods, ℓ_1-weighting modifies the iteration matrix to $B_{\ell_1} = (b_{i,j}^{\ell_1}) = I - (D + D^{\ell_1})^{-1}A$, where $D^{\ell_1} = (d_{i,i})$ is the diagonal matrix with entries

$$d_{i,i} := \text{sign}(a_{i,i}) \sum_{j \in \Omega_0^{(i)}} |a_{i,j}|.$$

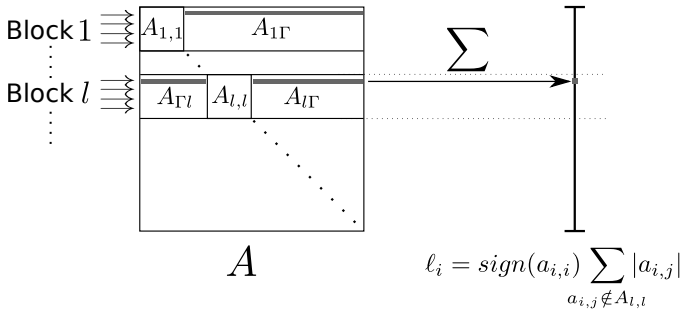


Figure 4.17.: Visualizing the ℓ_1-weighting technique.

Algorithm 20 Basic principle of using ℓ_1-weights in block-asynchronous iteration.

1: compute $B_{\ell_1} = (b_{i,j}^{\ell_1})$
2: update component i:
3: $s := d_i + \sum_{j \in \Omega_0^{(i)}} b_{i,j}^{\ell_1} x_j$ {off-diagonal part}
4: $x^{local} = x$
5: **for** $(k = 0;\ k < local\_iters;\ k++)$ **do**
6: $\quad x_i^{local} := s + \sum_{j \in \Omega^{(i)}} b_{i,j}^{\ell_1} x_j^{local}$ {using the local updates in the block}
7: **end for**
8: $x_i = x_i^{local}$

The obtained ℓ_1-weighted Block-Asynchronous Iteration can be found in Algorithm 20.

A question in this context is, whether the obtained weighted block-asynchronous iteration is still convergent.

4.10.2.4. Convergence of ℓ_1-weighted Block-Asynchronous Iteration

Theorem 4.10.2. *Suppose*

$$x_i^{k+1} = \sum_{j=1}^{n} b_{i,j}^{\ell_1} x_j^k + d_i, \quad i = 1, 2 \dots n \tag{4.5}$$

is an ℓ_1-weighted asynchronous iteration scheme where condition (a) and (b) are fulfilled. If furthermore

$$\rho\left(|B|\right) < 1$$

where $\rho\left(|B|\right)$ is the spectral radius of the component wise non-negative matrix B, then the sequence of solution approximations x^k is convergent to x^*, the unique solution of 2.1.

The proof to this is very similar to Theorem 3.2.2 with the difference, that the iteration matrix B of (2.20) is replaced by

$$B_{\ell_1} = \begin{pmatrix} 1 - \frac{a_{1,1}}{a_{1,1}+d_{1,1}} & -\frac{a_{1,2}}{a_{1,1}+d_{1,1}} & -\frac{a_{1,3}}{a_{1,1}+d_{1,1}} & \dots & -\frac{a_{1,n}}{a_{1,1}+d_{1,1}} \\ -\frac{a_{2,1}}{a_{2,2}+d_{2,2}} & 1 - \frac{a_{2,2}}{a_{2,2}+d_{2,2}} & \ddots & \vdots & \\ -\frac{a_{3,1}}{a_{3,3}} & \ddots & 1 - \frac{a_{3,3}}{a_{3,3}+d_{3,3}} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \\ -\frac{a_{n,1}}{a_{n,n}+d_{n,n}} & & \dots & & 1 - \frac{a_{n,n}}{a_{n,n}+d_{n,n}} \end{pmatrix}. \tag{4.6}$$

Introducing

$$R := \begin{pmatrix} \frac{a_{1,1}}{a_{1,1}+d_{1,1}} & 0 & 0 & \dots & 0 \\ 0 & \frac{a_{2,2}}{a_{2,2}+d_{2,2}} & 0 & & \vdots \\ 0 & 0 & \frac{a_{3,3}}{a_{3,3}+d_{3,3}} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \\ 0 & \dots & \dots & & \frac{a_{n,n}}{a_{n,n}+d_{n,n}} \end{pmatrix} \tag{4.7}$$

we can rewrite the iteration matrix into $B_{\ell_1} = R \cdot B + (I - R)$. Furthermore, due to the definition of $d_{i,i}^{\ell_1}$ we have $0 < r_{i,i} \leq 1 \; \forall i = 1 \dots n$.

Proof. To prove Theorem 4.10.2 we analyze like in the proof of Theorem 3.2.2 the difference between $x^* = 0$, the unique solution to (2.1) with $b = 0$ and the iterates x^k. Let $e^k = x^* - x^k$ again be the error in the t-th iteration. We consider the first $\bar{s}$ iterates in the process. Due to Theorem 3.2.3 we have that there exists a positive v and $\alpha < 1$ such that $|B|\, v \leq \alpha v$. Since all components of v are positive, there exists a positive value γ such that $|e^k| \leq \gamma v$ component wise for $0 \leq t \leq \bar{s}$. We now consider any component i that is updated using any of these $\bar{s}$ vectors forming the first $\bar{s}$ iterates. Without loss of generality we assumed $b = 0$ in (2.1) and obtain $d = 0$ in (4.5) and $B_{\ell_1} x_i^k = -B_{\ell_1} e_i^k$. As $0 < r_{i,i} \leq 1 \; \forall i = 1 \dots n$ the update satisfies

$$\begin{aligned} |e_i^{k+1}| &= \left| \sum_{j=1}^{n} b_{i,j}^{\ell_1} e_j^{k-s(t,j)} \right| \\ &= \sum_{j=1}^{n} \left| r_{i,i} b_{i,j} e_j^{k-s(t,j)} \right| + \left| (1 - r_{i,i}) e_i^{k-s(t,i)} \right| \\ &= |r_{i,i}| \sum_{j=1}^{n} \left| b_{i,j} e_j^{k-s(t,j)} \right| + |(1 - r_{i,i})| \left| e_i^{k-s(t,i)} \right| \\ &\leq \alpha |r_{i,i}| \gamma v_i + |(1 - r_{i,i})| \gamma v_i \\ &= \gamma \left(\alpha r_{i,i} + (1 - r_{i,i}) \right) v_i. \end{aligned}$$

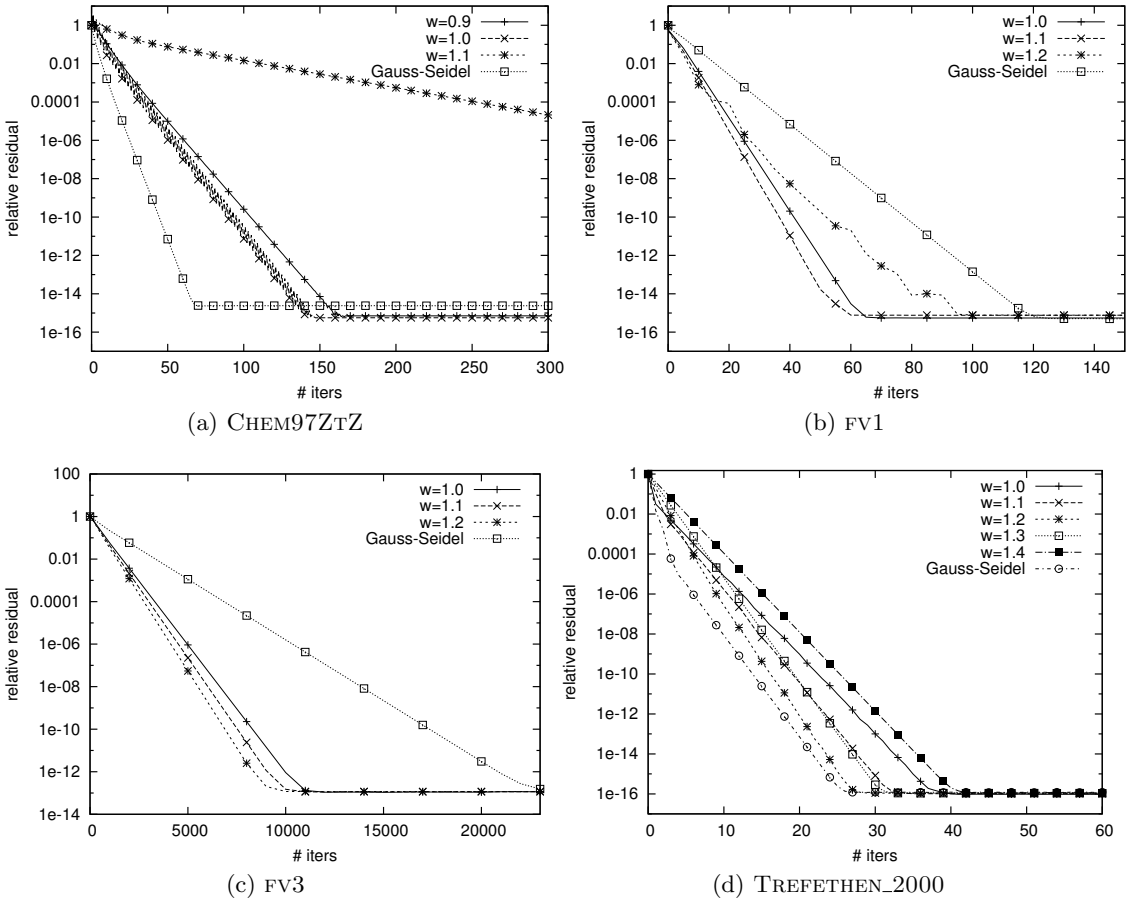
With $\beta_i = 1 - r_{i,i}(1 - \alpha)$ we have $0 \leq \beta_i < 1 \; \forall i = 1 \dots n$. Let $\beta = \max_i \{\beta_i\}$, then

$$|e_i^{k+1}| \leq \gamma \left(\alpha r_{i,i} + 1 - r_{i,i} \right) v_i \leq \beta \gamma v_i.$$

Now, we can continue like in the proof of Theorem 3.2.2. If t_1 is the first instance after $\bar{s}$ for which all components have been updated, then $|e^{k_1}| \leq \beta \gamma v$. Moreover, $|e^k| \leq \beta \gamma v$ for all $t \geq t_1$. Similarly, if t_2 is the next instance after t_1 for which all components have been updated a second time, then $|e_t| \leq \beta^2 \gamma v$ for all $t \geq t_2$. This way we obtain that the error $|e^k| = |x^* - x^k|$ converges to zero. Hence, the solution approximation is convergent to x^*, the unique solution. □

4.10.3. Experiments on Weighted Block-Asynchronous Iteration

While the experimental results presented in this section were presented at the HeteroPar 2012 workshop [ATDH12b], the properties and sparsity plots of the matrices we target can be found in Appendix B. Details about the hardware configuration of the used Disco-system can be found in Appendix C.3.



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Figure 4.18.: Convergence rate of ω-weighted block-asynchronous iteration for different choices of ω(=w) [ATDH12b].

On the CPU, the synchronous Gauss-Seidel and the SOR implementations run on 4 cores, which may come into account for the residual computation. Intel compiler 11.1.069 [int] is used with optimization flag "-O3". The GPU implementation is based on CUDA [NVI09], while the respective libraries used are from CUDA 4.0.17 [NVI11]. The component updates were coded in CUDA, using size of the matrix block decomposition was chosen according to the thread block size of 512.

In a first experiment, we analyze the influence of ω on the convergence rate with respect to global iteration numbers. Therefore we plot the relative residual depending on the iteration number. We want to stress again that all values are average due to the non-deterministic properties of block-asynchronous iteration. To have a reference, we additionally provide the convergence behavior of the sequential Gauss-Seidel algorithm.

The results reveal, that the convergence rate of the block-asynchronous iteration is very dependent on the matrix characteristics. For matrices with most relevant matrix entries gathered on or near the diagonal (small θ in (4.4)), the local Jacobi-like iterations provide sufficient update improvement to overcompensate for the inferior convergence rate of the block-asynchronous iteration conducted globally. Solving these systems, e.g. fv1 and fv3, we achieve a higher convergence rate than the sequential Gauss-Seidel algorithm. Similar to the SOR algorithm, choosing $\omega > 1$ may improve the convergence further (Figure 4.18b and 4.18c). Comparing the convergence rate of fv1 and fv3 we note that the condition number of the system has almost no influence.

For systems with significant off-diagonal entries, the convergence of the block-asynchronous

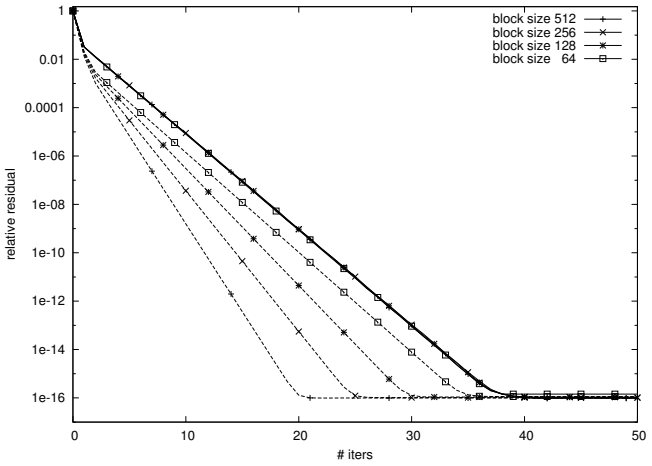


Figure 4.19.: Convergence improvement using ℓ_1-weights for different block sizes. Solid lines, all lying on top of each other, are unweighted block-asynchronous iteration, dashed lines are block-asynchronous iteration using ℓ_1-weights [ATDH12b].

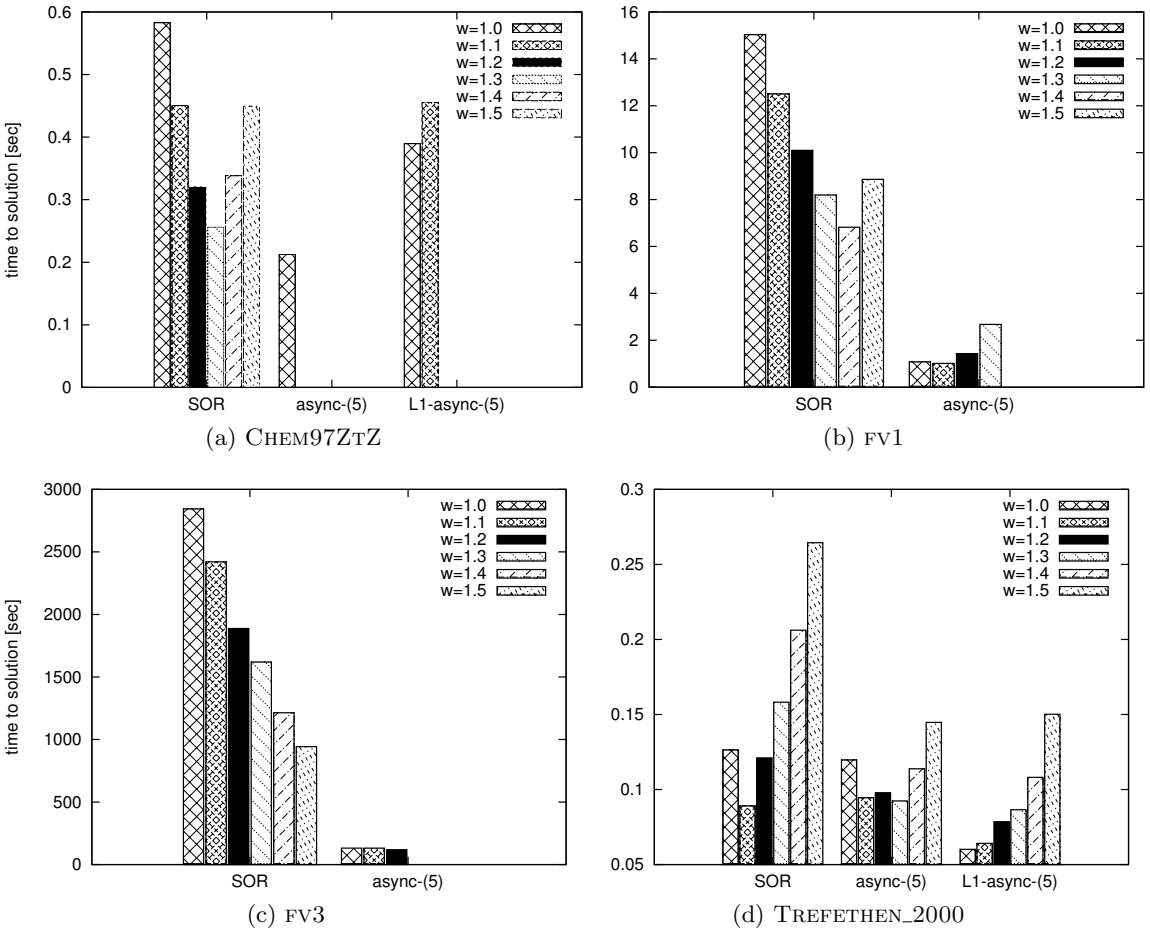
iteration decreases considerably (Figure 4.18a, 4.18d). The reason is that the off-diagonal entries are not taken into account for the local iterations. For these cases, it may not be beneficial to choose $\omega > 1$, eventually $\omega < 1$ would provide more improvement to the convergence rate.

The purpose of the ℓ_1-weights, introduced in Section 4.10.2.3, is to account for different off-diagonal parts in the distinct rows, i.e. to apply different weights in different rows. We now want to evaluate whether this approach triggers convergence improvement. For the tests it is reasonable to choose a system with a high number of off-diagonal elements. Therefore, we will focus our analysis on the matrix TREFETHEN_2000 where, due to the design, the ratio between the diagonal entry and the offdiagonal parts differs considerably for the distinct rows. We now compare, using different block sizes, the convergence rate of the block-asynchronous iteration with the ℓ_1-weighted variant. Note at this point that using ℓ_1-weights triggers some overhead to the computation time due to the computation of the weights. This may be daunting for some systems where the convergence improvement is smaller than the overhead, but as computing the weights is relatively cheap, the improved convergence rate may in most cases directly correlate to a performance increase.

We can observe in Figure 4.19 that, independently of the block size, using ℓ_1-weights improves the convergence rate. We also note that the influence of the block-size on the convergence rate for the unweighted algorithm is negligible. Furthermore, using ℓ_1-weights is especially beneficial when targeting large block sizes, where the ℓ_1-weights for the distinct rows in one block differ considerably. For this case (e.g. block size 512), the convergence of the block-asynchronous iteration is improved by a factor of almost 2 compared to the unweighted algorithm.

While the convergence rate, with respect to iteration number, is interesting from the theoretical point of view, the more relevant factor is the time-to-solution performance. This depends not only on the convergence rate, but also on the efficiency of the respective algorithm on the available hardware resources (hardware-dependent iteration rate). Whereas the Gauss-Seidel algorithm and the derived SOR algorithms require strict update order and hence only allow sequential implementations, block-asynchronous iteration is very tolerant to update order and synchronization latencies, and therefore adequate for GPU implementations.

In the next experiment, we analyze the runtime performance for the ω-weighted block-



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Figure 4.20.: Time-to-solution comparison between SOR and weighted block-asynchronous iteration [ATDH12b].

asynchronous iteration, and compare it with the SOR algorithm. We want to remind that despite the similar notation, ω-weighting has, due to the algorithm design, a very different meaning in the SOR and the block-asynchronous iteration, respectively. For reasonable test cases, i.e. for the matrices with considerable off-diagonal entries (large θ in (4.4)), we provide additional data for different ω-weights applied to the block-asynchronous algorithm enhanced by the ℓ_1-weighting technique.

The results show that for matrices where most entries are clustered on or near the main diagonal, the ω-weighted block asynchronous iteration outperforms the SOR method by more than an order of magnitude, see Figure 4.20b and 4.20c. We also observe, that ω-weights for the block-asynchronous algorithm have to be applied more carefully: already choosing $\omega \geq 1.4$ leads to divergence of all test cases. For matrices with considerable off-diagonal parts, using the block-asynchronous iteration may not pay off when comparing with SOR.

Considering the runtime analysis for the matrix Chem97ZtZ (Figure 4.20a) we have to realize that although the unweighted block-asynchronous iteration generates the solution faster than SOR, using ω-weights is not beneficial. While already choosing $\omega = 1.1$ results in the loss of convergence, choosing $\omega \in [1, 1.1)$ may have positive effects. The algorithm also does not benefit from enhancing it by ℓ_1-weights, which may stem from the very unique matrix properties. We notice however that, despite the poor performance, ℓ_1-weights have positive impact on the algorithm's stability: for $\omega = 1.1$, the convergence of the block-asynchronous iteration is maintained.

For the test matrix TREFETHEN_2000, the performance of SOR and block-asynchronous iteration is comparable for ω near 1. But enhancing the latter one with ℓ_1-weights triggers considerable performance increase. We then outperform the SOR algorithm by a factor of nearly 5 (see Figure 4.20d). This reveals not only that using ℓ_1-weights pays off, but also the potential of applying a combination of both weighting techniques.

4.10.4. θ_l-dependent Block-Asynchronous Iteration

A different approach to account for the off-block entries can be realized by adapting the number of local iterations on the subdomains to the respective diagonal dominance. The idea is to decrease the local iteration count where the off-block parts are large compared to the block parts, and residual improvement via iterating locally is expected to be small. Conversely, the number of local iterations may be increased where the most significant entries are clustered in the block diagonal.

The result is a block-asynchronous algorithm, where considerable differences can be expected in the number of updates for the distinct components. Nevertheless, the convergence rate may be improved compared to the block-asynchronous method where all components are handled similarly. To investigate this issue, we compare in Section 4.10.5 the convergence rate of the async-(5) method with the async-(θ_l) variant, where the number of local iterations is adapted to the ratio between the on- and off-block entries. The notation async-(θ_l) is chosen due to the linear system decomposition introduced in Section 4.10.2. While θ was used in (4.4) to quantify the off-block part per row, we now denote the number of local component updates in block i with θ_l. The following steps may be used to determine the respective iteration count θ_l:

- For each row, the sum of the absolute values of the block elements and of the off-block elements is computed.
- For each block, as well as for the complete matrix, the average ratio between the block and off-block parts is calculated.
- For blocks with a ratio higher than the average of the complete matrix, the number of local component updates is increased. Conversely, for blocks with a ratio lower, the number of local component updates is decreased.
- The sum of all local component updates forming one global iteration is computed. In case of a higher total iteration count compared to the original algorithm, the number of local updates is decreased for all blocks. This ensures, that the total number of component updates in one global iteration is not larger than before, since this would falsify the comparison with the original code.

As a result, in the obtained θ_l-dependent block-asynchronous iteration, the total iteration count in one global update is equal or smaller to the unweighted block-asynchronous iteration with fixed number of local component updates, but some (block-) components may be processed more often.

The performance of the obtained θ_l-dependent asynchronous iteration could be improved further, if the components in the block with less local iterations are scheduled more often such that more global updates are conducted for them. We refrain from this idea since it requires complex modifications in the scheduling mechanisms of the used hardware.

4.10.5. Experiments on θ_l-dependent Block-Asynchronous Iteration

To analyze the convergence behavior of async-(θ_l), where the number of local iterations is determined by the ratio between the elements located in the block diagonal and the

block size	# blocks	# local iters	
		async-(θ_l)	async-(5)
28	72	318	355
56	36	157	175
112	18	76	85
224	9	37	40
448	5	16	20

Table 4.9.: Local Iteration Numbers for θ_l-dependent Block-Asynchronous Iteration applied to TREFETHEN_20000. The last columns report the total number of component updates in one global iteration.

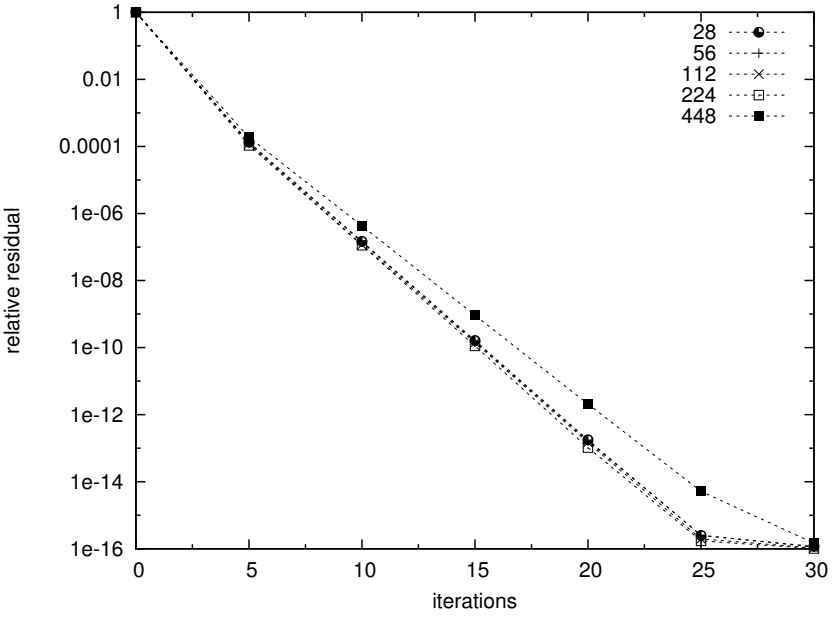


Figure 4.21.: Convergence rate of async-(θ_l) for different block sizes.

off-block elements, we focus on the system of linear equations associated with the matrix TREFETHEN_2000 (Appendix B). The size and structure of this test case allows detailed comparison for different block sizes.

We have noticed in Figure 4.19 in Section 4.10.3 that the block size has almost no impact on the convergence rate for the unweighted async-(5) algorithm. Hence, it is not surprising to observe similar properties for the async-(θ_l) method reported in Figure 4.21. Nevertheless, one difference to the async-(5) convergence can be identified: the convergence is faster for smaller block sizes. This stems from the fact that smaller blocksizes allow for a finer adaptation of the local iterations. Comparing the convergence of async-(θ_l) and async-(5) in Figure 4.22 we realize that using the θ_l-adapted algorithm improves the convergence by about 30% - despite the fact that the total number of local iterations per global iteration was reduced by more than 13% (Table 4.9). This result shows the superiority of async-(θ_l) over async-(5) for this test case. In general, replacing the block-asynchronous iteration method using a fixed number of local component updates by async-(θ_l) is only beneficial if the overhead associated with the computation of the θ_l is compensated for by faster convergence.

An interesting observation can be made when comparing with the ℓ_1-weighted async-(5) method. As we have seen in Section 4.10.3, the convergence behavior of the weighted

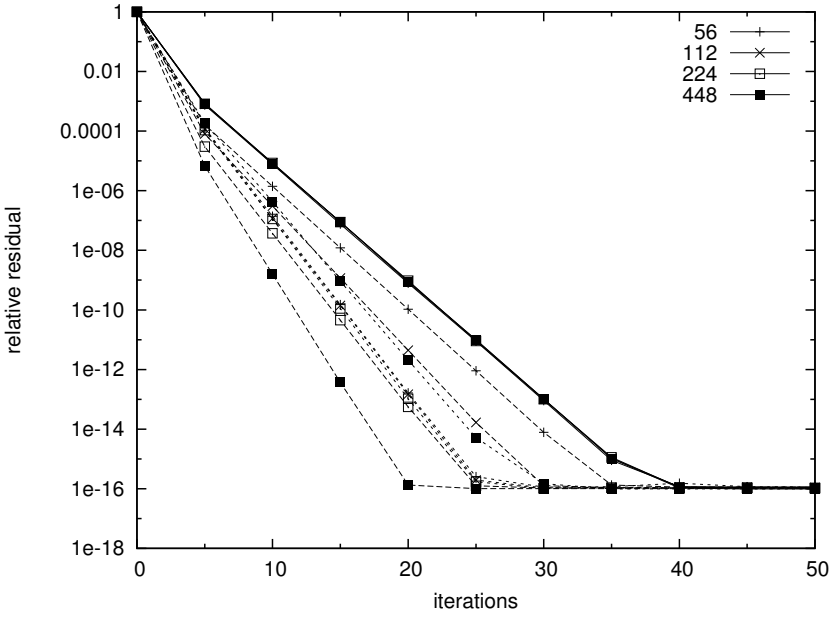


Figure 4.22.: Convergence rate comparison between different block-asynchronous implementations for different block sizes: solid lines are async-(5) (all lying on top of each other), dashed lines are ℓ_1-async-(5) and dotted lines (all close to each other) are async-(θ_l).

method is influenced by the block size (see Figure 4.19). In the comparison with the async-(θ_l) iteration we observe that the ℓ_1-weighted variant converges faster for large block sizes, and slower for small block sizes. Hence, depending on the hardware system configuration and the problem, async-(θ_l) may be an efficient alternative to the ℓ_1-weighted variant.

4.10.6. Block-Asynchronous Iteration for PDE-Discretizations

As we have seen in the previous sections, using weights in block-asynchronous iteration may improve the convergence rate and the runtime performance. This was achieved by introducing matrix weights or adapting parameters like the number of local iterations and the block size to a specific problem. While optimizing the parameters was based on the matrix properties, the performance on a specific hardware platform also depends on the system architecture like the number of computing cores, the cache sizes etc.

A very convenient situation occurs if the system of linear equations is taken from a finite element discretization of an elliptic partial differential equation. Then, it may be possible to increase the convergence rate of the block-asynchronous iteration considerably by adapting to the discretization mesh. This stems from the fact, that choosing a block size equivalent to the number of elements/nodes in a certain direction of the finite element or finite difference discretization will reduce the off-diagonal parts in the block decomposition of the component matrix significantly. This principle also serves as basis for the concept of line relaxation techniques in synchronous two-stage iterations [Saa03]. In order to illustrate the idea, we dedicate the following section to adapting the block size to a finite difference discretization of the two-dimensional Helmholtz equation on the unit square. Although this example may seem very simple, the underlying idea stays the same when targeting more complex problems.

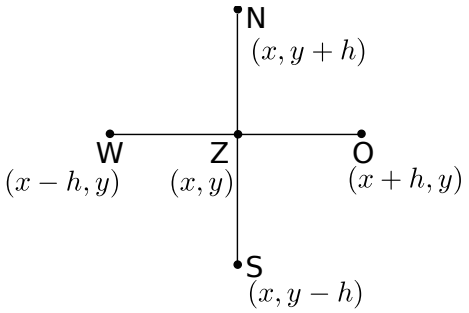


Figure 4.23.: Five-point-Stencil.

4.10.7. Block-Asynchronous Iteration adapted to 2D Helmholtz

The Helmholtz Equation is an Elliptic partial differential equation of second order that can for the two-dimensional case with Dirichlet[1] boundary conditions [Bra07] be written as

$$-\Delta u + \delta u = b \quad u \in \Omega, \tag{4.8}$$
$$u = 0 \quad u \in \partial\Omega, \tag{4.9}$$

where $u : \Omega \to \mathbb{R}$ and $\Omega \subset \mathbb{R}^2$ [Bra07, GT01]. Recall that $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$.

Assuming a unit square, a possible finite difference discretization can be obtained by the five-point stencil [Bra07]. In this case, for approximating the solution at any grid point, we need the values of the four adjacent points. For a boundary point, we only need the values of the neighbors that are included in the domain. Usually, the values for the nodes on the boundary are anyway defined by the boundary conditions (in our case Dirichlet boundary conditions (4.9)). The resulting five point stencil for the inner points (visualized in Figure 4.23), becomes

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4+\delta & -1 \\ & -1 & \end{bmatrix}$$

where h is the uniform distance between two neighboring nodes.

For the two-dimensional unit square, we choose an equidistant and uniform $(N+2)\times(N+2)$ discretization (x_i, y_j), $i, j \in \{0, 1 \ldots N, N+1\}$. Then,

$$\begin{aligned} x_i &= i \cdot h, \\ y_i &= i \cdot h, \\ h &= \frac{1}{N+1}, \\ n &= N^2. \end{aligned}$$

For example, using and a lexicographic numbering of the nodes [Bra07], the system of

[1] Peter Gustav Lejeune Dirichlet (⋆1805, †1859)

linear equations (4.8), (4.9) can be discretized for $N = 3$ in the form:

$$\frac{1}{h^2}\left(\begin{array}{ccc|ccc|ccc} 4+\delta & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4+\delta & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4+\delta & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & 4+\delta & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4+\delta & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4+\delta & 0 & 0 & -1 \\ \hline 0 & 0 & 0 & -1 & 0 & 0 & 4+\delta & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4+\delta & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4+\delta \end{array}\right) \cdot x = b. \quad (4.10)$$

We observe, that this system can be rewritten into the block system

$$\frac{1}{h^2}\begin{pmatrix} H & -I & 0 \\ -I & H & -I \\ 0 & -I & H \end{pmatrix} \cdot x = b, \quad (4.11)$$

where

$$H = \begin{pmatrix} 4+\delta & -1 & 0 \\ -1 & 4+\delta & -1 \\ 0 & -1 & 4+\delta \end{pmatrix}, \quad I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad 0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Choosing the block size equivalent to the number of inner (free) nodes in one direction of the discretization, i.e. choosing the block size equal to three for $N = 3$ with fixed boundary, we obtain a block asynchronous algorithm where every diagonal block has at most two off-block entries in every row. Obviously, choosing any other block size (except for naive block size of the system's dimension n) we would get a higher number of elements not being part of any diagonal block. Since these elements are not taken into account in the local iterations on the matrix diagonal blocks, the convergence is potentially improved by reducing the number of off-diagonal elements.

In the following section we run experiments on a large system of linear equations to show how adapting the block size to the Finite Difference Discretization improves the convergence rate. At this point, we also want to mention, that similar methods can be applied when targeting domain decomposition methods in the discretization of partial differential equations. The block size may then be adapted to the number of elements inside one subdomain, and solving the local problem corresponds to the local iterations while the coupling via the boundary corresponds to the global iterations [Bra07].

4.10.8. Experiments for Block-Asynchronous Iteration adapted to PDE Discretizations

For the Five-Point Stencil introduced in the last section, we investigate the trade off between the block size and the number of nodes per direction. In Table 4.10 the characteristics of the respectively arising systems of linear equations can be found for different discretizations. In a first test, we use a discretization with $N = 32$, where the partial differential equation and the boundary values are given by (4.8, 4.9) with $\delta = 10^{-2}$. To the resulting system of linear equations with $n = 1024$ unknowns we apply the block asynchronous iteration using different block sizes, and report in Figure 4.24 the respective average residual after 1000 iterations.

According to our expectations, a larger block size triggers faster convergence. This stems from the fact, that a larger block size captures more row elements in the local iterations.

mesh $(N+2 \times N+2)$	N	n	# nnz	est. condition number
102×102	100	10000	49600	670.98
103×103	101	10201	50601	673.11
104×104	102	10404	51612	675.20
105×105	103	10609	52633	677.23
106×106	104	10816	53664	679.22
107×107	105	11025	54705	681.17
108×108	106	11236	55756	683.07
109×109	107	11449	56817	684.93
110×110	108	11664	57888	686.75
111×111	109	11881	58969	688.52
112×112	110	12100	60060	690.26
113×113	111	12321	61161	691.96
114×114	112	12544	62272	693.63
115×115	113	12769	63393	695.25
116×116	114	12996	64524	696.85
117×117	115	13225	65665	698.41
118×118	116	13456	66816	699.93
119×119	117	13689	67977	701.43
120×120	118	13924	69148	702.89
121×121	119	14161	70329	704.32
122×122	120	14400	71520	705.72

Table 4.10.: System Characteristics for the finite difference discretization of the Helmholtz Equation (4.8) with Dirichlet boundary conditions (4.9) and δ in (4.8) is set to 10^{-2}. Due to the fixed boundary values, the number of unknowns per direction (N) is two smaller than the number of elements per direction.

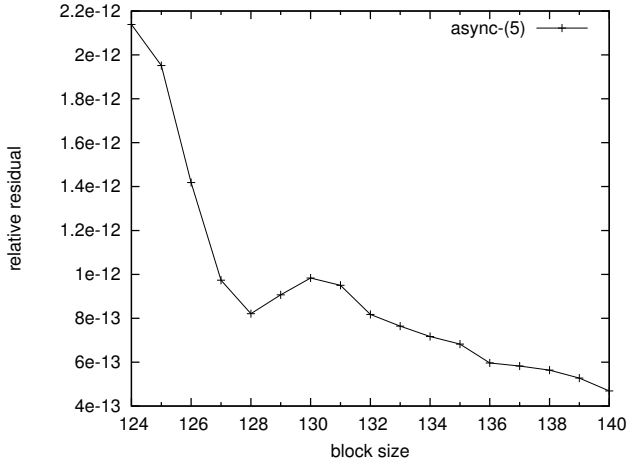


Figure 4.24.: Average relative residual after 1000 iterations of async-(5) for different block sizes applied to the Finite Difference Discretization of 2D Laplace on a equidistant grid with given boundary values and $N = 32$.

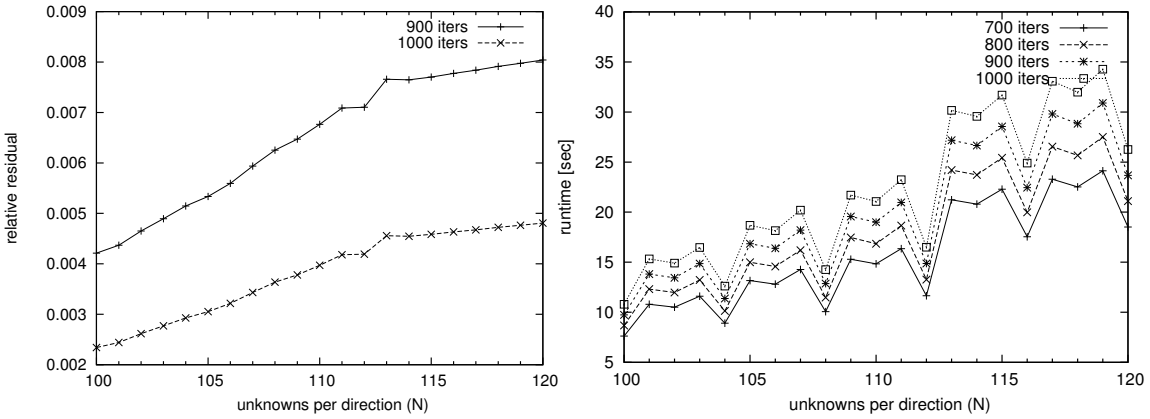
But the convergence improvement does not linearly correlate to the block size: We observe a local minimum for the block size 128, which happens to be a multiple of the number of (free) elements per direction ($N = 32$). Not only is the convergence considerably faster than for smaller block sizes, but also faster than for the block sizes 129, 130, 131, 132. This confirms our assumption that convergence can be improved when choosing the blocksize adapted to the number of unknowns per direction, since it lowers the number of components that are not taken into account in any of the local iterations. Hence, it may be beneficial to adapt the block size of the implementation to the discretization of the partial differential equation.

On the other hand, the computational power of the used devices varies considerably when implementing different block sizes due to the hardware design. Especially for block sizes adapted to the physical core number, significant performance improvement can be expected. For this reason, one might argue whether working the other way round, i.e. adapting the discretization method to the used hardware, may be beneficial too. This question is not straight-forward, since choosing a different discretization mesh usually impacts the accuracy and performance of the solution process. For example, choosing a lower number of nodes per direction usually triggers higher discretization error [Bra07]. Due to accuracy requirements it may be reasonable to exclude this case from further investigation.

A higher numbers of nodes per direction on the other hand leads to a higher number of unknowns. Additionally, for partial differential equations including derivatives, choosing a finer discretization usually also increases the condition number. (We can observe this effect in Table 4.10.) Both effects, the larger number of unknowns and the higher condition number, may result in slower convergence of the iterative solver. Hence, choosing a higher number of nodes per direction one would in general expect to have a slower convergence and an extended time-to-solution. Only if the interaction between linear system, solver implementation and hardware performance compensate for the higher condition number and the higher number of unknowns, we can hope for improved convergence and time-to-solution performance.

To investigate this issue, we report in Figure 4.25a the average residual after 900 and after 1000 iterations of async-(5) for the Finite Difference Discretization of the Helmholtz Equation (4.8) with Dirichlet boundary condition (4.9) for different meshes. While we stick to the unit square domain, a fixed block-size of the matrix decomposition of 112 and set $\delta = 10^{-2}$ for all tests, we increase the number of nodes per direction, which impacts the associated system of linear equations (see Table 4.10). Analysing the data, we observe an almost linear dependency between the relative residual after 900/1000 iterations and the element number per discretization direction - except for the mesh with $N = 112$ unknowns per direction. For this case, the relative residual is smaller than expected according to the overall trend. At the same time, we observe significant variations in the algorithm's performance for different sizes of the discretized system. In Figure 4.25b we can identify a pattern and realize that choosing the mesh adapted to the block size is beneficial to the performance.

To target the question of how these effects combine, we report in Figure 4.26 the convergence with respect to runtime. The results show that choosing the number of nodes adapted to the block size, the algorithm's runtime-dependent convergence is faster than for coarser grids. A solution approximation of a certain accuracy for $N = 112$ is achieved in shorter runtime than for discretizations using $N = 111, 110, 109, 107, 106$ or $N = 105$, only for $N = 108$, the runtime-based convergence is superior. This is quite astonishing, since refining a discretization with $N = 105$ by adding seven more nodes per direction not only results in a system with 1519 more unknowns (about 10%), but also in a higher condition number (see Table 4.10). The more efficient hardware usage compensates for



(a) Average Relative Residual (b) Runtime

Figure 4.25.: Average relative residual and solver runtime for different iteration numbers of async-(5) using a fixed block size of 112 applied to different discretizations of the 2D Laplace.

the additional computational cost, and refining the coarser grid is beneficial in this case. We want to stress again, that this refinement even comes with the advantage of a smaller discretization error.

A conclusion of this section is, that not only adapting the implementation with respect to the problem, but also choosing the discretization with respect to hardware and implementation is a necessary step to achieve high performance.

4.11. Asynchronous Iteration Smoothers in Multigrid Methods

4.11.1. Multigrid Smoothers

As we have seen in Section 2.6, one critical component of multigrid methods is the smoother, since it is crucial for the convergence rate as well as the runtime performance of the multigrid solver. Usually, a simple relaxation method such as Gauss-Seidel or Jacobi is used for pre- and post-smoothing the solution approximations on the distinct grid levels. From the analytical point of view, if the error is expressed in terms of the eigenvectors of the system, the smoother must reduce the error components associated with the eigenvectors having large eigenvalues, while the coarse-grid correction eliminates the remaining error contribution [Tro00].

The typically applied smoothers, such as Gauss-Seidel, usually do not parallelize well (see 2.4.2). Therefore, much effort is put into developing parallel smoothers that scale on multicore architectures. A possible approach is to use a set of local smoothers that exchange boundary values in a Jacobi-like manner [HY00, AMB06]. The performance of these hybrid smoothers may then be enhanced furthermore by using weights [MY11].

Still, the synchronization necessary to exchange boundary values may be detrimental to the performance on highly parallel architectures. Therefore, it may be worthwhile to consider a block-asynchronous iteration for the smoother in multigrid methods as it lacks any synchronization and therefore scales optimally on any architecture. In the following sections we introduce the mathematical problem we target and report the experimental results obtained by comparing block-asynchronous smoothers with Gauss-Seidel smoothers. Most of following material is taken from [ATG+12].

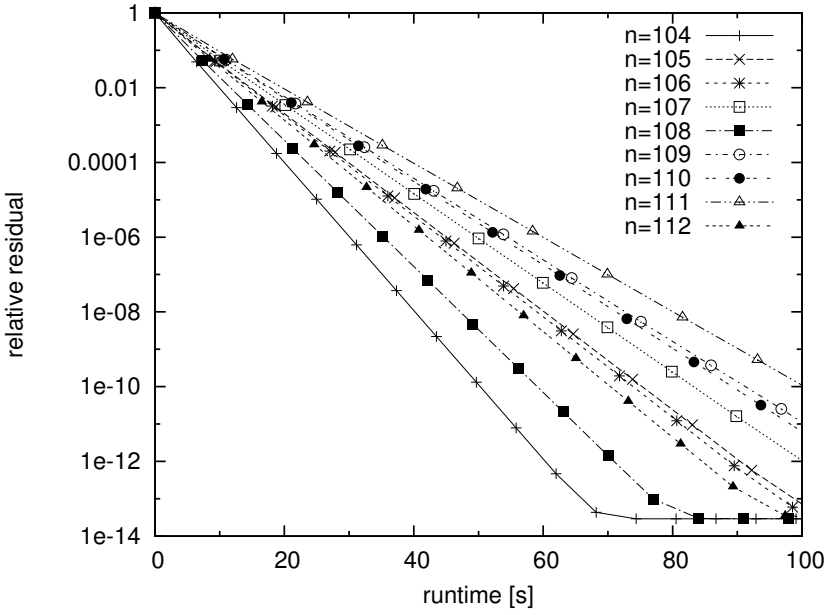


Figure 4.26.: Convergence of async-(5) using a block size of 128 for different discretizations. The displayed number of unknowns per direction (N) is two smaller than the number of nodes per direction in the discretization.

4.11.2. Numerical Experiments on Block-Asynchronous Smoothers

4.11.2.1. Experimental Setup

The numerical problem we target is the finite difference discretization of the differential equation

$$-\Delta u + \varepsilon u = f,$$

where $u : \Omega \to \mathbb{R}$ and $\Omega \subset \mathbb{R}$. For Dirichlet boundary conditions equal to zero, the 1D discretization for this problem on a grid of size h can be written as a system of linear equations of the form $Ax = b$ with

$$A = \begin{pmatrix} 2+h^2\varepsilon & -1 & 0 & \dots & 0 \\ -1 & 2+h^2\varepsilon & \ddots & \ddots & \vdots \\ 0 & \ddots & 2+h^2\varepsilon & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & 0 & -1 & 2+h^2\varepsilon \end{pmatrix}. \tag{4.12}$$

Although this may seem to be a very basic problem, it contains many essential aspects necessary to analyze the convergence behavior of the multigrid method. It can be shown that the condition number of the matrix A can be estimated by $\kappa = 4 \cdot \frac{1}{h^2\varepsilon}$ [Var04]. In the considered experiments, we set $h = 1$ and vary $\varepsilon \in [10^{-6},\ 10^{-1}]$ in order to investigate the influence of the condition number on the solver performance.

The geometric multigrid method we apply to this system is implemented according to Algorithm 10 (Section 2.6), where we use the Conjugate Gradient method for the solution of the coarse grid system (see Section 2.5.3). To analyze the performance of a GPU-based block-asynchronous iteration as smoother, we compare it with a CPU implementation of Gauss-Seidel performing smoothing iteration.

For all smoothers, we use a stencil implementation of the corresponding iteration method, updating the distinct components by using the adjacent components [Saa03, RHMDR07,

AHW09]. This reduces the computational cost, since we do not have to perform a sparse matrix vector multiplication, as well as the memory requirements, which are usually daunting when performing GPU-based kernels. We want to mention at this point, that the use of stencils instead of a matrix system is only possible due to the problem's characteristics, it is impossible when targeting irregular grids or local refinement. Still, we utilize the explicit matrix to compute the error term on each grid level. Utilizing stencils for this may be beneficial for the overall performance as well, but we refrain from doing so since this is not the main target of the analysis in this thesis.

In general, it is challenging to analyze the performance of a smoother within a multigrid framework. The reason is that the smoothers' properties depend on tunable parameters, e.g., related to the linear system's characteristics, the applied multigrid scheme, the solver used on the coarsest grid, etc. We therefore split the numerical tests into two parts, where we first analyze the two-grid iteration and then extend it to a complete multilevel V-cycle.

We can identify three parameters that can be used to adjust the smoother. These are the number of global iterations that correspond to the number of iterations of a synchronous iterative method like Jacobi or Gauss-Seidel, the number of local iterations on the subdomains, and the size of the subdomains. The number of global iterations is usually dominating the execution time of a block-asynchronous iteration on GPUs (see Algorithm 14), which is still small compared to synchronous Gauss-Seidel on the CPU (see Section 4.4). We have seen in Section 4.4.2 that adding local iterations on the GPU, due to the data locality and the GPU architecture, almost comes for free [ATDH12a]. But at the same time, adding local iterations may not trigger the same improvement to the solution approximation. Since in general the factor between the convergence rate of Gauss-Seidel and Jacobi equals to two, we always merge two global block-asynchronous iterations into one smoothing step. The local iterations may then be used to compensate for the convergence loss due to the chaotic behavior of the asynchronous method. Without investigating the trade-off between global and local iterations, we set the latter one like before to the fixed number of five. In the second part of the numerical experiment section we then extend the Two-Grid iteration to a full V-cycle. We analyze the impact of adding grid levels, and report the smoother run times for different problem sizes. Finally, we provide a detailed time-to-solution comparison between block-asynchronous iteration and Gauss-Seidel smoothed multigrid for a 10-level implementation using different numbers of smoothing steps.

4.11.2.2. Numerical Experiments

The experiments were conducted on the Supermicro system (see Appendix C.1). For the CPU parts of the multigrid method, 4 cores on the same CPU are used. Due to the inherently sequential Gauss-Seidel implementation, only the grid operations, such as restriction and prolongation, can leverage this parallelism provided by the CPU cores. The Intel compiler version 11.1.069 [int] is applied with optimization flag "–O3". The GPU implementations of block-asynchronous iteration and Jacobi are based on CUDA [NVI09], while the respective libraries used are from CUDA 4.0.17 [NVI11]. The component updates were coded in CUDA, using thread blocks of size 512.

In the first experiment, we analyze the impact of the condition number of the system of linear equations on the performance of multigrid methods smoothed by block-asynchronous iteration, Jacobi and Gauss-Seidel. We choose a dimension of $n = 10,000,000$ and compare the convergence with respect to the iterations for different condition numbers κ. Figure 4.27a shows numerical results using one smoothing step of Gauss-Seidel, Jacobi or block-asynchronous iteration, while the results in Figure 4.27b are for two smoothing steps. First, we observe that the number of necessary multigrid steps to convergence can be considerably decreased by performing two instead of one smoothing iteration. Second, the Jacobi

smoother is not able to provide similar smoothing improvement: the convergence of the Jacobi-smoothed method is considerably slower than for the other methods. Furthermore, the block-asynchronous smoother has smoothing properties similar to the Gauss-Seidel, so the convergence behavior of the multigrid is almost not affected when replacing Gauss Seidel by async-(5). For very small condition numbers, the block-asynchronous iteration performs in many cases even better than the Gauss-Seidel smoother. The only difference is the accuracy of the final solution: The Gauss-Seidel method allows a higher approximation quality than the block-asynchronous iteration. But the variations are small and the more crucial factor determining the accuracy of the final solution approximation are the limitations of the floating point format. Still, if very accurate solution approximations are requested, it may also be reasonable to switch to a Gauss-Seidel method for the last V-cycles.

Motivated by the results of these experiments we refrain from including the Jacobi smoother in the further comparison. The considerably higher number of V-cycles necessary to converge cannot be compensated for by the parallelism of the Jacobi method.

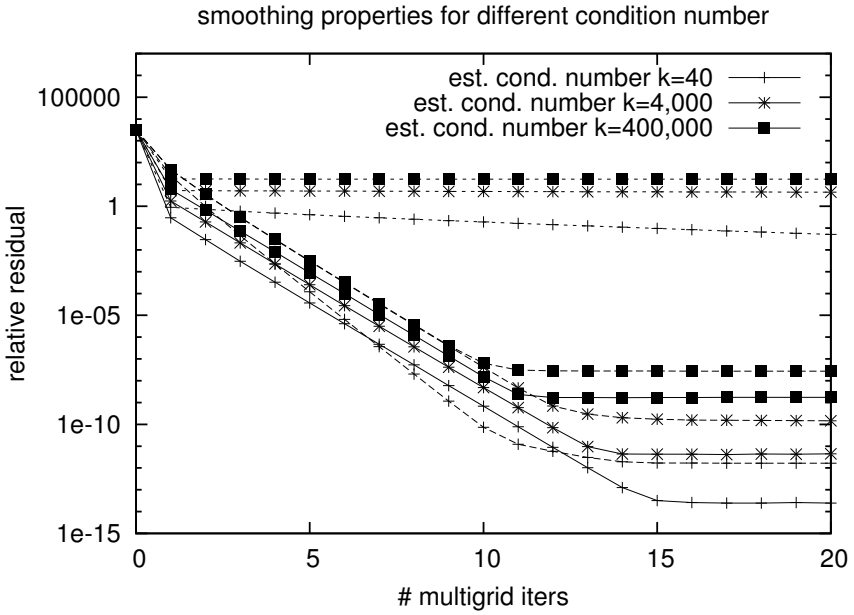
In the next experiment we investigate the impact of the problem size on the finest grid level. For this purpose we choose problem sizes between 10^4 and 10^8 and analyze the convergence behavior. The results shown in Figure 4.28 reveal that the problem size has almost no influence on the convergence rate.

Like in previous experiments the convergence rate with respect to iteration number is interesting from the theoretical point of view, the more relevant factor is the convergence with respect to time. In Table 4.11 and Figure 4.29, we report run times of the respective smoothers for different problem sizes. We also extend the analysis from a two-grid method to multiple levels.

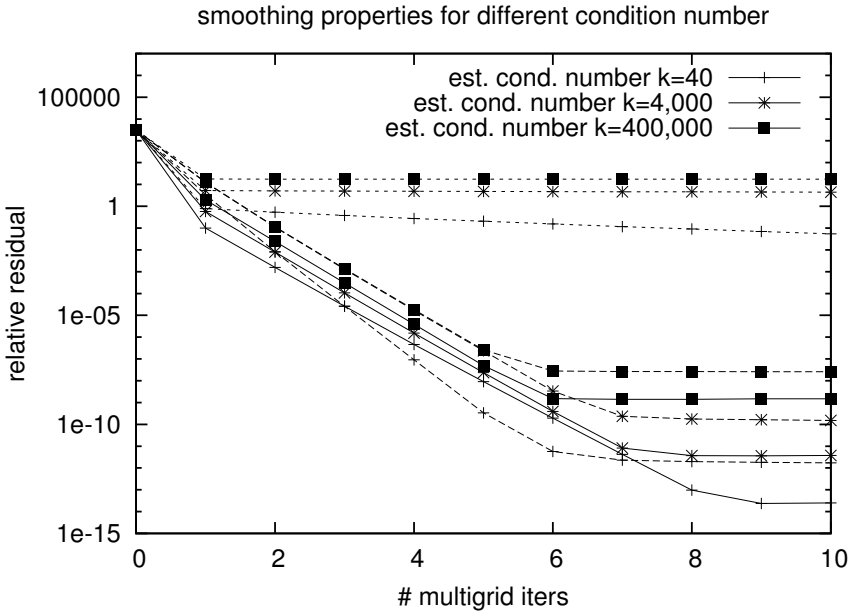
The run times are the aggregated smoother times for the multigrid to converge, which is the point where the approximation accuracy does no longer benefit from adding V-cycles. Since we usually obtain higher accuracy approximations for the Gauss-Seidel smoother, we choose a stopping criterion for the multigrid iteration that can be achieved for both methods. Additionally we provide the data transfer time for the GPU implementation of the block-asynchronous iteration smoother. This also contains the overhead of the GPU initialization. Note, that we report only the run times for the smoother, which increase for multiple levels due to additional smoother calls. The total runtime for the multigrid iteration may still decrease with more levels due to the smaller system of linear equations solved on the lowest grid level.

We observe in Table 4.11 that for Gauss-Seidel on the CPU, increasing the problem size on the finest grid corresponds to a linear run time increase for the smoother. This is also true for the block-asynchronous iteration on the GPU, except for small problem sizes, where calling the GPU kernels triggers some overhead. Calling the GPU-based smoother for the first time also includes the GPU initialization. For all problem sizes and grid sequences, the async-(5) smoother outperforms the Gauss-Seidel smoother. While for small problems the improvement is at least a factor of three, it rises to 7 for larger dimensions. Since the multigrid framework is in the majority of cases implemented on the host of the system, we should also take the data transfer time into account. Then, for small problem sizes, the async-(5) smoother suffers from this overhead due to the GPU initialization and expensive data transfer. This effect can be observed in Figure 4.29 where we visualized the runtime results of Table 4.11 for the case of a two- and five-level multigrid method. For larger problem sizes, the async-(5) smoother outperforms the Gauss-Seidel smoother at least by a factor of two, independent of the number of grid levels.

The question is how this corresponds to an acceleration of the multigrid method, since the smoother only accounts for one part in the overall execution time beside the grid



(a) One Pre-/Post-Smoothing Step



(b) Two Pre-/Post-Smoothing Steps

Figure 4.27.: Two-level convergence for $n = 10,000,000$ and different condition numbers using one or two smoothing steps, respectively. Dashed lines are block-asynchronous, dotted lines are Jacobi, and solid lines are Gauss-Seidel, [ATG+12].

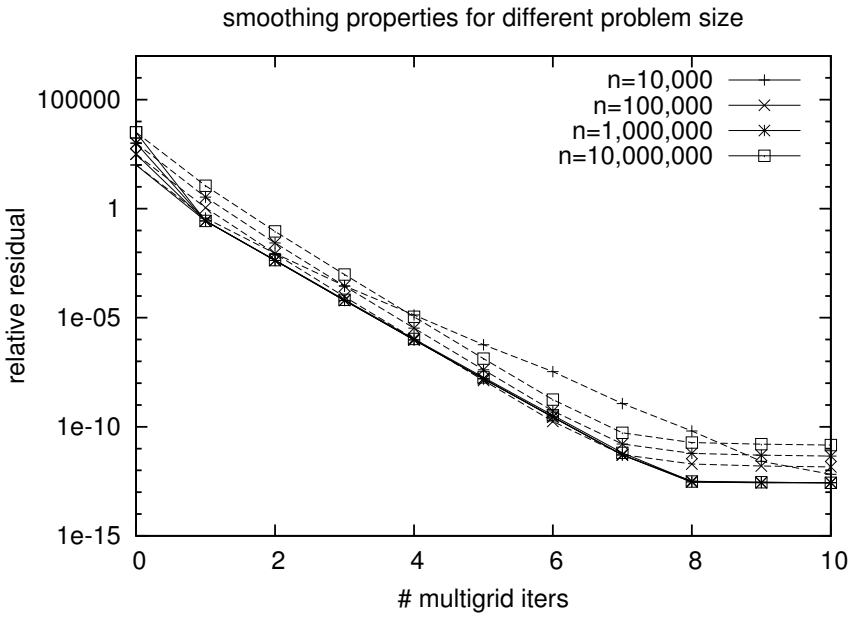


Figure 4.28.: Two-level convergence using two smoothing step for different problem sizes. Dashed lines are block-asynchronous and solid lines, all lying on top of each other, are Gauss-Seidel, [ATG+12].

operations and the coarse grid solver [Tro00]. (Still, the smoother may dominate the overall multigrid performance in a variety of cases, see e.g. [GPT07].) To investigate this issue, we apply a 10-level multigrid method to problems of size 10,000,000 and different condition numbers, and provide detailed analysis on the execution time of the smoother, the grid operations like restriction, prolongation and residual computation, and the direct solver on the coarsest grid level.

Analyzing the results in Figure 4.30, we realize that applying more smoothing steps reduces the number of V-cycles in the multigrid method, which again reduces the number of solver calls on the coarsest grid level and the number of grid operations. This effect can easily be seen in the runtime of the grid operations which directly corresponds to the number of multigrid iterations. Considering the trade-off between V-cycles and smoothing steps, there is a point for maximal performance. In Figure 4.30a we identify the maximal performance when conducting one smoothing step for Gauss-Seidel, respectively six smoothing steps for async-(5). For higher condition numbers it seems beneficial to conduct more smoothing steps: We then achieve optimal performance for two, respectively five smoothing steps (see Figure 4.30b). The number of smoothing steps minimizing the overall execution time can in general be determined only heuristically. As already seen, in our case, it differs not only for different condition numbers, where more smoothing steps are beneficial for higher condition numbers, but also between the block-asynchronous smoother and the Gauss-Seidel smoother (Table 4.11). The reason is that the block-asynchronous iteration is not only considerably faster than Gauss-Seidel, but is also dominated by the data transfers between host and CPU. The component updates using a stencil for the block-asynchronous iteration come almost for free. Therefore, increasing the number of iterations does not cause a linear increase of the computation time. Hence, for large numbers of smoothing steps, the speedup factor between Gauss-Seidel and the block-asynchronous iteration smoothed multigrid rises. Comparing the results for smaller and larger condition numbers (see Figure 4.30a and Figure 4.30b, respectively) one might conclude that higher condition numbers (at least for our test cases) may have minor impact on the multigrid performance. This may be true for the time-to solution, but neglects the fact, that the accuracy of the solution approximation depends on the condition number and is lower for

	dimension:	10e+3	10e+5	10e+6	10e+7k
2 levels	Gauss-Seidel	0.00398	0.03613	0.36815	3.62690
	async-(5)	0.00085	0.00602	0.05397	0.53082
	transfer	0.01193	0.02571	0.10779	0.85057
3 levels	Gauss-Seidel	0.00621	0.05539	0.54973	5.60806
	async-(5)	0.00181	0.00913	0.08433	0.80394
	transfer	0.02243	0.04267	0.16522	1.29404
4 levels	Gauss-Seidel	0.00805	0.06621	0.64393	6.48827
	async-(5)	0.00242	0.01044	0.09654	0.92531
	transfer	0.03289	0.05673	0.20734	1.53345
5 levels	Gauss-Seidel	0.00957	0.07181	0.69662	6.95943
	async-(5)	0.00338	0.01156	0.10531	1.00858
	transfer	0.04102	0.06866	0.23634	1.64843

Table 4.11.: Average smoother runtime [s] for different numbers of levels performing always 2 Pre- and 2 Post-smoothing steps on the respective grid levels. We report the timings for Gauss-Seidel on CPU, block-asynchronous iteration (async-(5)) on GPU and the data transfer time between host and GPU, [ATG+12].

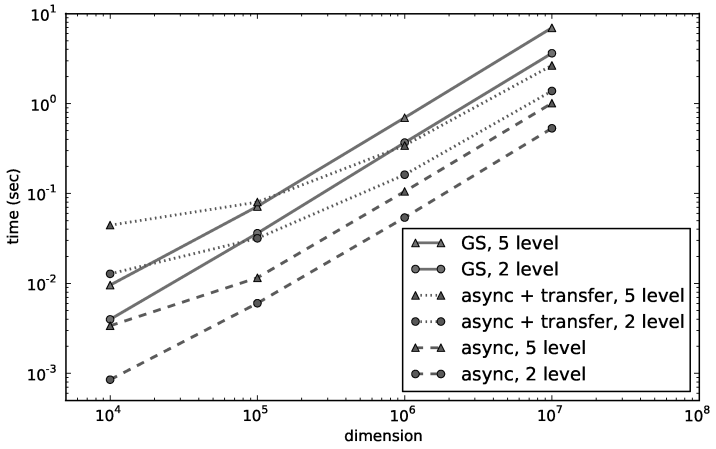


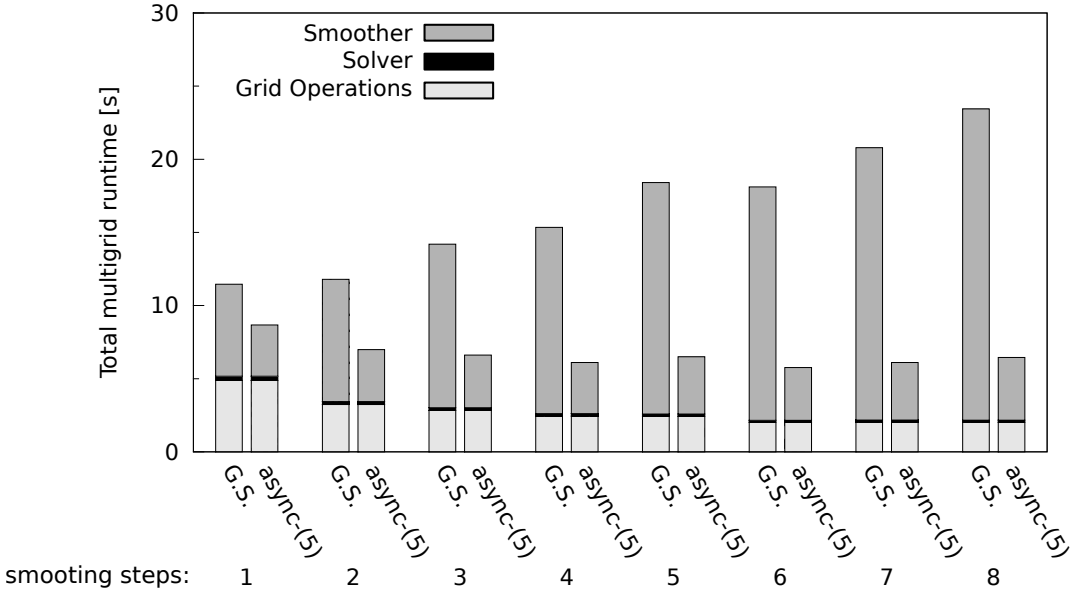
Figure 4.29.: Average smoother runtime for the two- and five-level multigrid method using Gauss-Seidel (GS) or async-(5) smoother with two Pre- and 2 Post-smoothing steps on the respective grid levels, [ATG+12].

ill-conditioned systems.

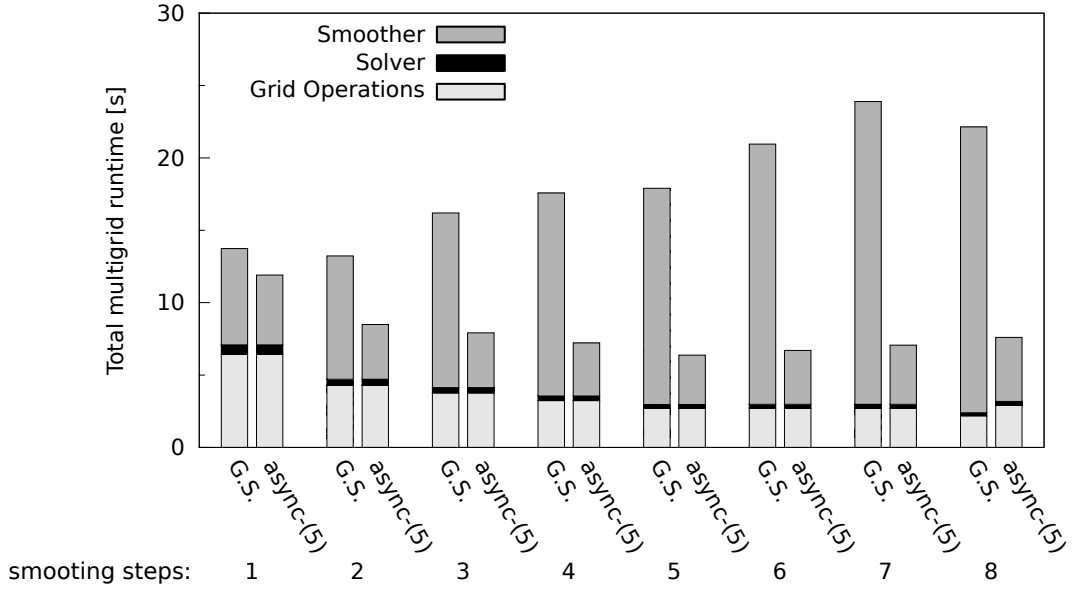
Since the smoother accounts in our experiments for a high percentage of the overall multigrid time, replacing the Gauss-Seidel smoother leads to considerable acceleration. While for more complex problems, the ratio between smoother and overall multigrid time is often smaller since the coarse grid solver becomes more expensive, the necessity of more smoothing steps rises at the same time, making the block-asynchronous smoother even more attractive. Note that this analysis also takes the data transfer time into account. Implementing the multigrid framework on the GPU like proposed in [FL08, GSMy+08, GT08, Tho08] could result in even higher speedups.

4.12. Block-Asynchronous Error Correction in Mixed Precision Iterative Refinement

In the last sections, we have seen that asynchronous methods may have the potential to outperform their synchronous counterparts in terms of performance. This is achieved by



(a) Problem size n=10, 000, 000, condition number estimate $\kappa \approx 10^3$.



(b) Problem size n=10, 000, 000, condition number estimate $\kappa \approx 10^6$.

Figure 4.30.: 10-level multigrid V-cycle runtime analysis for different numbers of Pre- and Post-smoothing steps using Gauss-Seidel and async-(5), respectively. The async-(5) smoother includes data transfer times to and from the GPU, [ATG+12].

the efficient leverage of the computation power provided by the hardware. The absence of synchronization points enables to transfer the floating point performance of the used hardware to the algorithm. In Section 2.3 we already introduced mixed precision iterative refinement, another well-known technique used to leverage the potential of accelerators. The basic idea there was to use a lower precision format for the error correction solver inside an iterative refinement method at full precision. An open question is how a combination of mixed precision iterative refinement and block-asynchronous iteration impacts the convergence and properties and the performance. On the one hand the methods share the same principle for gaining their performance: they both compensate their low conver-

gence properties by leveraging the high computational power of GPUs [ALDH12]. But on the other hand, combining them is a challenge since the iterative refinement introduces synchronization points that we try to avoid in asynchronous iteration.

4.12.1. Numerical Experiments on Block-Asynchronous Error Correction in Mixed Precision Iterative Refinement

The experimental setup and the results in this section are mostly according to the conference contribution [ALDH12] where the combination of the two methods is analyzed for different hardware architectures. While we base the GPU implementations of the block-asynchronous iteration on CUDA [NVI09], the respective libraries used are from CUDA 2.3 for the C1060 and the GTX280, and CUDA 4.0.17 [NVI11] for the C2070 and GTX580 implementation. The kernels updating the respective components, launched through different streams, use thread blocks of size 512, see Section 4.1. The thread block size, the number of streams, along with other parameters, were determined through empirically based tuning. Details about the used hardware can be found in Table C.4 in the Appendix. For the iterative refinement implementation we use a first outer iteration to analyze the residual improvement and then adapt the number of inner iterations such that the residual improvement equals the accuracy of floating point precision in every outer update. Hence, while the first error correction loop may provide different improvements for the distinct test cases, the further loops all decrease the residual by 6 to 8 digits. In case of the mixed precision implementations, the error correction solver is implemented using single precision. Due to the low precision representation of the system of linear equations, additional rounding errors may be expected, slowing down the convergence of the iterative refinement. To analyze this issue, we compare in a first experiment the convergence behavior of the iterative refinement method using a double- and a single- precision error correction solver, respectively. Using different precision formats, the vectors and the linear system have to be converted from double to single precision. This typecast, handled by the GPU, triggers some overhead and may be crucial for problems where only very few iterations of the error correction solver are executed. To analyze the impact of the overhead of iterative refinement and the use of different precision formats, we will also provide the solver runtimes for the different systems of linear equations for the plain block-asynchronous iteration in double precision, the iterative refinement in double precision and the mixed precision iterative refinement, where the latter ones use the block-asynchronous iteration as an error correction solver.

In the first experiment, we analyze how using lower precision for the block-asynchronous iteration error correction solver impacts the iterative refinement convergence rate. Therefore, we report the relative residual depending on the iteration number for chosen systems of linear equations given in Appendix B. Note again, that due to the implementation, the first outer loop is used to determine the residual improvement, while the further iterations improve the approximation iterate by 6 to 8 digits, depending on the rounding error.

The results in Figure 4.31 show that for the test matrices Trefethen_2000, Chem97ZtZ and fv1, using single precision for the error correction solver has a nearly negligible impact on the convergence of the iterative refinement. Only for the fv3 test case, the convergence rate is affected. This was expected since the high condition number of this matrix (see Appendix B) may cause representation errors in the low precision format that make the approximation updates less beneficial.

Like before, the convergence behavior is only interesting from the theoretical point of view, and the next experiment is dedicated to the analyzing how handling the error correction equation in single precision impacts the performance. The motivation is that using single

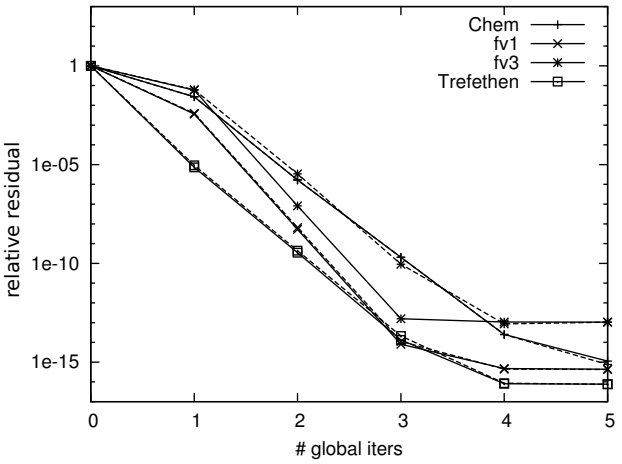


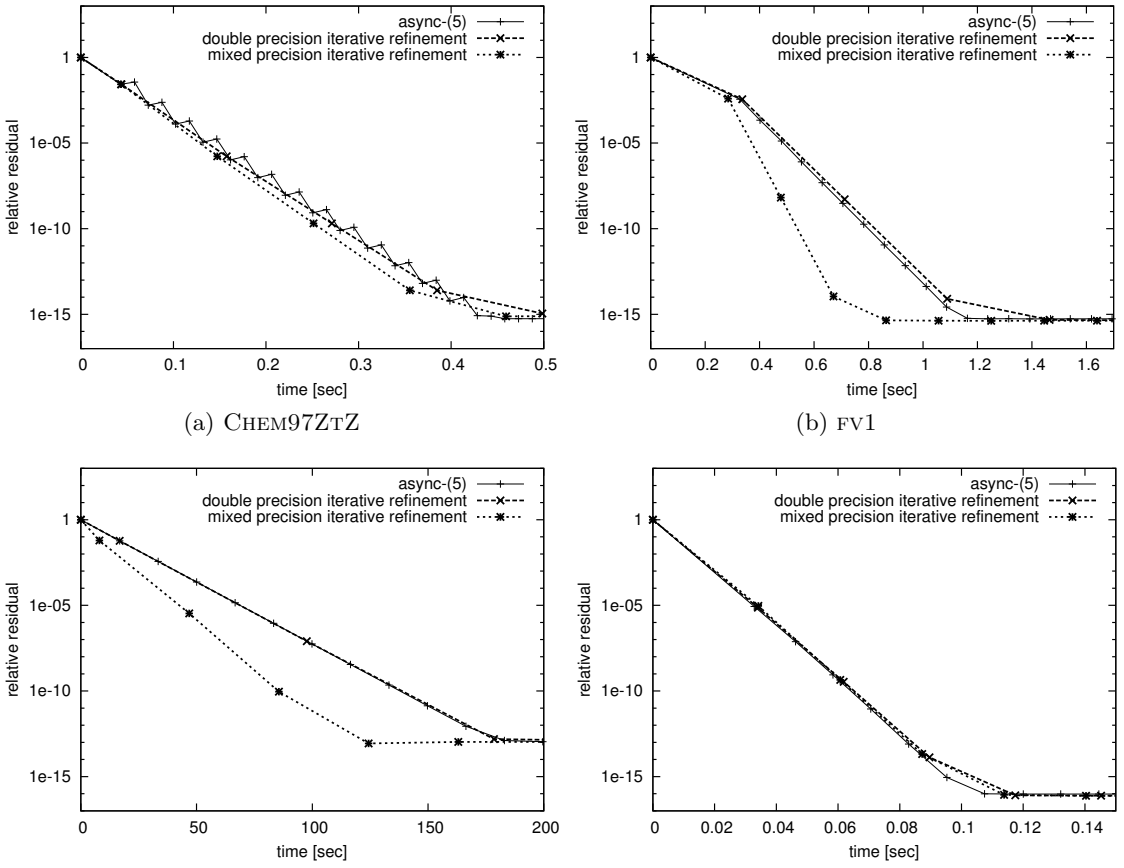
Figure 4.31.: Iterative refinement convergence, solid lines are double-precision error correction, dashed lines are single-precision error correction.

instead of double as working precision, should trigger some speedup. From the computational point of view, switching from double to single precision should generate a speedup of two, but on some hardware devices, even larger differences between single and double precision performance can be observed, see e.g. the specifications of the C1060 GPU in C.4. Also a sophisticated memory hierarchy allowing for the efficient caching and prefetching sometimes allows for speedups that extend this expected value. See Appendix A.2 for a short summary on this issue. Depending on the hardware setup and the specific problem, these speedups may potentially overcompensate for the overhead associated with the typecast between the formats.

While the convergence, with respect to iteration number, is independent of the hardware used, the performance depends on the architecture. We use the C2070 for this experiment, as this 'Fermi' generation is in 2012 the state of the art from the Nvidia GPU manufacturer. In addition to the convergence performance of the iterative refinement, using a double or single precision error correction solver, we report the results for the plain block-asynchronous iteration in double precision.

In 4.32 we observe, that for all test cases, the overhead is negligible when embedding the block-asynchronous iteration (async-(5)) in double precision into the iterative refinement framework. For the small test cases CHEM97ZTZ and TREFETHEN_2000 (Figure 4.32a, 4.32d), switching to the mixed precision iterative refinement approach gives no improvement. For the larger matrices, e.g. FV1, the improvement by using low precision for the error correction solver is relevant (Figure 4.32b): The mixed precision implementation converges in almost half the computation time than the double precision implementation. Even for the test case FV3, where we observed a slower convergence rate for the mixed precision approach in Figure 4.31, we benefit in terms of performance (Figure 4.32c).

We may now target different hardware platforms, and report in Figure 4.33 the respective time-to-solution. For the test cases FV1 and FV3, despite the performance difference between single and double precision of around 10 on the C1070 and GTX280 (see Table C.4), the mixed precision iterative refinement performs inferior to the plain double implementation of async-(5). The reason is, that for these systems, the GPU-internal memory bandwidth is the limiting factor and the overhead, due to the iterative refinement framework, can not be compensated for by the single precision performance. For the small matrices, things are different. Since the size of CHEM97ZTZ and TREFETHEN_2000 allows for the caching of the iteration vector as well as the right-hand side, the C1060 and the GTX280 are able to leverage the single precision performance more efficiently. Still, the



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

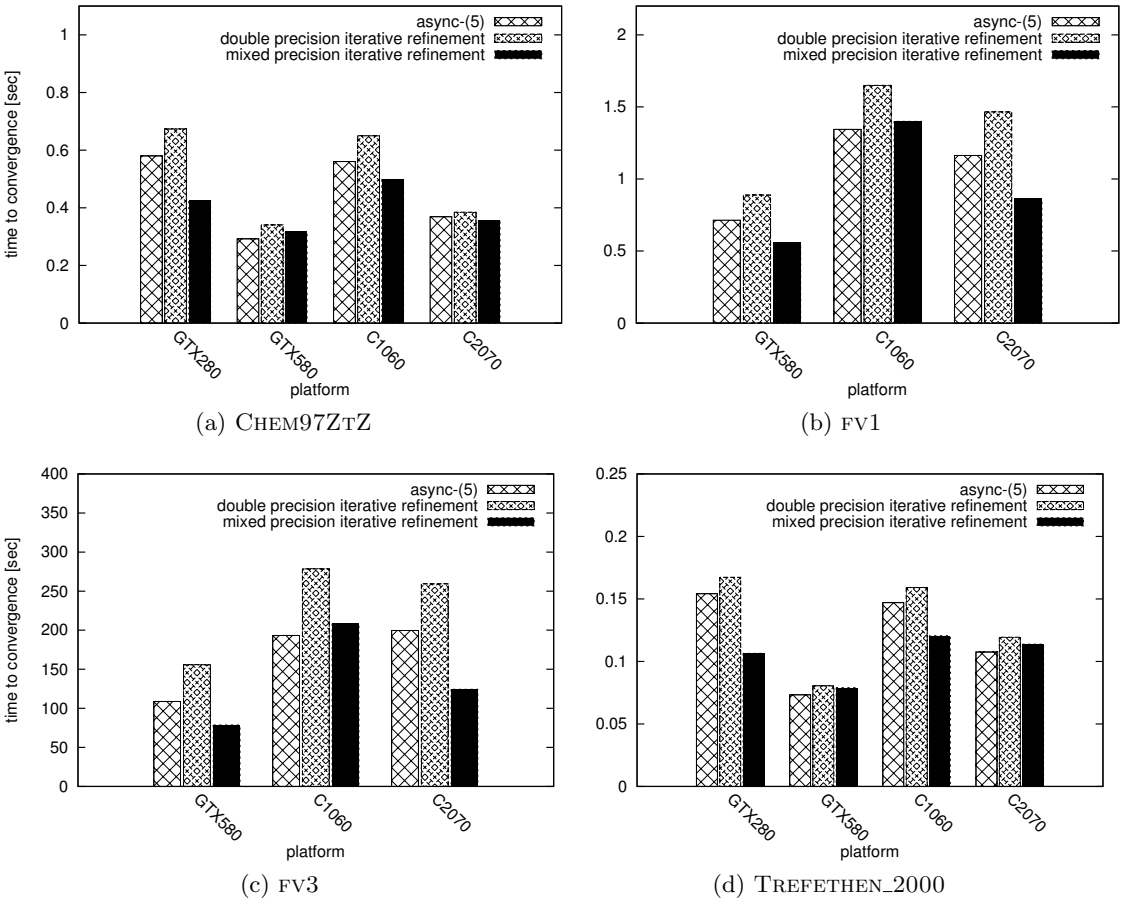
Figure 4.32.: Iterative refinement performance, time-dependent relative residual.

bandwidth remains the limiting factor, since the complete matrix cannot be loaded into cache, and the higher memory bandwidth of the consumer version (GTX280) explains the better performance for the mixed precision approach. Using double precision, the server version (C1060) is superior, probably due to the more sophisticated memory structure. Unfortunately, the very limited main memory on the GTX280 does not allow for the handling of large systems.

Note that the total solver runtime of the mixed precision approach for Trefethen_2000 is on the GTX280 even smaller than on the server version of the Fermi line (C2070). An explanation may be that the overall runtime also includes the initialization process, which has to be taken into account for this system, and may be longer for GPUs using CUDA in version 4.0 and equipped with more memory.

Targeting the Fermi generation, we observe that, especially for large systems, we benefit from the mixed precision framework. Although we may only expect a factor of two concerning the floating point performance, the sophisticated memory hierarchy may enable even higher speedups. This speedup stems from the fact that, not only are we able to keep the iteration vector and the right-hand side local due to the larger L1 cache, but also because the L2 cache allows for the efficient data access of the iteration matrix.

We want to mention that for the test case fv3, the iterative refinement in double precision fulfills the outer residual stopping criterion after 4 iterations, while we could observe in Figure 4.31 that it is already very close after 3 iterations. Hence, the double precision iterative refinement runtime would benefit from choosing a smaller number of inner iterations for the last global iteration. Nevertheless, we expect this runtime reduction to be small



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Figure 4.33.: Total solver runtime.

and impacting the performance order in the direct comparison to the block-asynchronous implementation.

4.13. Asynchronous Iterative Refinement

In Section 2.2 we introduced iterative refinement methods, that use the residual of a computed solution as right-hand side to solve an error correction equation. Using different precision formats within the distinct parts of the iterative algorithm, we obtained the mixed precision iterative refinement method (see Section 2.3). The flexibility in terms of choosing the error correction solver allows for employing asynchronous relaxation methods. In Section 4.12 we discussed the challenge of combining block-asynchronous iteration with (mixed precision) iterative refinement. The key point is the trade-off between synchronizations induced by the iterative refinement and the asynchronism of the block-asynchronous iteration. While we aim for reducing the synchronizations to leverage the computing capabilities of the hardware, the iterative refinement framework needs synchronization to compute the residuals between the solution updates. Nevertheless, the results in Section 4.12.1 reveal the potential of combining block-asynchronous iteration with mixed precision iterative refinement.

Now we want to step further, by allowing for asynchronism not only in the defect correction solver, but also in the iterative refinement framework itself.

Before targeting this challenge we recall that the iterative refinement operator $F(x)$ is

given by

$$x^{k+1} = F(x^k) = x^k + c^k, \tag{4.13}$$

where c^k denotes the solution update computed by the error correction solver applied to $Ac^k = r^k$ with the residual $r^k = b - Ax^k$. We may write this update in the form $c^k = sol(A^{-1}r^k) = sol(A^{-1}(b - Ax^k))$, where $sol(\cdot)$ denotes the (approximate) solution of the error correction equation obtained by applying any (direct or iterative) error correction solver.

In case of using a direct error correction solver and exact arithmetic, we would have

$$sol(A^{-1}(b - Ax^k)) = A^{-1}b - A^{-1}Ax^k$$

and

$$x^{k+1} = F(x^k) = x^k + sol(A^{-1}(b - Ax^k)) = x^k + A^{-1}b - x^k = A^{-1}b,$$

which implies that applying one step of iterative refinement would provide the exact solution. Furthermore Nicholas J. Higham [Hig96] provides a rule of thumb stating that iterative refinement for Gaussian elimination produces a solution correct to working precision if double the working precision is used in the computation of the residual r, e.g. by using quad or double extended precision IEEE 754 floating point (see Appendix A.1), and if A is not too ill-conditioned [Hig96].

Implementations however usually contain rounding effects due to limited precision, and therefore

$$sol(A^{-1}(b - Ax^k)) \approx A^{-1}b - A^{-1}Ax^k.$$

This is especially true when using iterative error correction solvers iterating to a certain residual accuracy, like we do in Section 4.12. To account for the approximation error when using an iterative error correction solver and/or limited floating point precision, we introduce the approximation error $\varepsilon(A, b, x^k)$

$$sol(A^{-1}(b - Ax^k)) = A^{-1}b - A^{-1}Ax^k + \varepsilon(A, b, x^k). \tag{4.14}$$

This approximation error depends on the characteristics of the error correction equation $Ac^{k+1} = b - Ax^k$, the stopping criterion of the iterative solver and rounding effects in case of limited precision. Note that in case of employing an error correction solver using lower precision than working precision, like we do in Section 2.3), the approximation error additionally depends on the representation errors in the low precision format as the error correction solver is applied to a perturbed system.

Deriving an asynchronous iterative refinement, we have to consider the convergence properties for the obtained algorithm. Since it is difficult to write the iteration operator F in the form $F(x) = Bx + d$ in case of using iterative error correction, the convergence theory for asynchronous methods given in Section 3.2.1 can hardly be applied. In [Bau78] Gérard M. Baudet provides more general convergence results for asynchronous methods based on contracting operators. In the following, we recall the most important aspects.

Definition 4.13.1. *[Bau78] An operator $F : \mathbb{R}^n \to \mathbb{R}^n$ is a Lipchitzian operator on a subset $D \subset \mathbb{R}^n$ if there exists a nonnegative matrix L such that*

$$|F(x) - F(y)| \leq L|x - y|, \quad \forall x, y \in D,$$

componentwise. The matrix A is called Lipchitzian matrix for the operator F.

Definition 4.13.2. *[Bau78] An operator $F : \mathbb{R}^n \to \mathbb{R}^n$ is a contracting operator on a subset $D \subset \mathbb{R}^n$ if it is a Lichitzian operator on D with the spectral radius of its Lipchitzian matrix L fulfilling $\rho(L) < 1$.*

Theorem 4.13.3. *[Bau78] If F is a contracting operator on a closed subset $D \subset \mathbb{R}$ and if $F(D) \subset D$, then the asynchronous iteration obtained from F and starting with $x^0 \in D$ converges for any update pattern to the unique fixed point of F in D.*

Proof. The proof follows the same concept like the proof of Theorem 3.2.3 with the difference that we may now consider the Lipchitzian instead of the iteration matrix. See [Bau78] for more details. □

To ensure convergence for asynchronous iterative refinement, we need F of (4.13) to be a contracting operator. For this purpose we analyze $|F(x) - F(y)|$.

4.13.1. Convergence of Asynchronous Iterative Refinement using Exact Floating Point Arithmetic and an Exact Error Correction Solver

First, we consider the academic case of exact arithmetic and an exact error correction solver. For this case, $\varepsilon(A, b, x^k) = 0$ in (4.14) and

$$\begin{aligned} |F(x) - F(y)| &= \left|x + sol(A^{-1}(b - Ax)) - y - sol(A^{-1}(b - Ay))\right| \\ &= \left|(x - y) + A^{-1}(b - Ax) - A^{-1}(b - Ay)\right| = 0. \end{aligned} \tag{4.15}$$

Hence, the Lipchitzian matrix $L = 0$, and F is a contraction. We deduce from Theorem 4.13.3 that the asynchronous iterative refinement converges for any initial guess and any update pattern. However, this result is mainly of theoretical value, since computing an error correction term component wise may be difficult using an exact solver. Also, implementations based on floating point arithmetic with limited precision usually introduce rounding errors.

4.13.2. Convergence of Asynchronous Iterative Refinement using an Iterative Error Correction Solver and/or Limited Floating Point Precision

As already mentioned, the conditions of Section 4.13.1 are rarely fulfilled. Floating point arithmetic implies rounding effects, and asynchronous iterative refinement suggests to use component wise iterative error correction solvers. Hence, we may have $\varepsilon(A, b, x^k) \neq 0$ in for the solution updates in the different steps of (4.14) and derive

$$\begin{aligned} |F(x) - F(y)| &= \left|x + sol(A^{-1}(b - Ax)) - y - sol(A^{-1}(b - Ay))\right| \\ &= \left|(x - y) + A^{-1}(b - Ax) + \varepsilon(A, b, x) - A^{-1}(b - Ay) - \varepsilon(A, b, y)\right| \\ &= |\varepsilon(A, b, x) - \varepsilon(A, b, y)| \end{aligned} \tag{4.16}$$

We realize that the contraction property of the iterative refinement operator F depends on the approximation quality of the error correction solver. In case of a sufficient approximation accuracy, we may have $|\varepsilon(A, b, x) - \varepsilon(A, b, y)| \leq L|x - y|$ with $L < 1$. However, it is very difficult to provide general statements about the approximation error and whether F is a contraction. This stems from the fact that the accuracy of the iterative error correction solver, the representation errors and the rounding effects for every single component may have significant impact on this property. Especially when using a low precision iterative error correction solver where the error correction solver is applied to perturbed

problem, the dependencies are very complex. In [Hig96] Nicholas Higham goes the burdensome path to derive a theory on the contraction property of mixed precision iterative refinement. Therefore he assumes the perturbations associated with solving the low precision error correction to be bounded by some operator depending on the linear system characteristics. Without restating the theory in detail we use his theory providing the contraction property for the mixed precision iterative refinement operator F. If now the asynchronous iterative refinement method based on F furthermore starts with $x^0 \in D$ ($F(D) \subset D$, closed subset), we obtain by applying Theorem 4.13.3 that the sequence of iterates converges to the unique fixed point of F in D for any update pattern.

4.13.3. Block-Asynchronous Iterative Refinement

Algorithm 21 Asynchronous Iterative Refinement.

for all ($i \in \{1 \dots n\}$) **do** {asynchronous outer loop}
  read x from global memory
  set $y = x$
  compute residual in component $r_i = b_i - (Ay)_i$
  compute solution update c_i from $Ac = r$
  update $y_i = x_i + c_i$
  overwrite x_i in global memory with y_i
end for

Analyzing the asynchronous iterative refinement in Algorithm 21, we can identify the computation of the solution update c_i for component i as a critical component. This has several reasons. First, we note that the right-hand side of the error correction equation changes permanently due to the asynchronously computed residuals. Second, computing an error correction term for one component only is very difficult in general. Using the complete error correction system to compute the demanded component update is inefficient, since the updates for all other components would be lost. Using all of them on the other hand brings us full circle back to the synchronized iterative refinement. For these reasons, when running iterative refinement component wise asynchronously, the only reasonable error correction solvers are the asynchronous relaxation methods we introduced in Chapter 3.1. Another possibility is to split the linear system into blocks, and to apply a *Block-Asynchronous Iterative Refinement*. The approximation updates for the components located in the same block can then be computed using a (synchronous or asynchronous) relaxation method on the sub-matrix, while the different blocks are processed asynchronously. This approach is very similar to using block-asynchronous iteration as error correction solver inside (synchronous) iterative refinement (Section 4.12) with the difference, that the iterative refinement process is no longer synchronized. For this reason the algorithm can also be derived by applying the computing blocks of block-asynchronous iteration not to the original linear system of equations (2.1), but to a residual system which is constantly changing. Particularly, the residuals for the components in a block are generated on-the-fly as error correction equation just before starting the iteration process. In Algorithm 22 we propose a possible implementation using a relaxation method $K(x, b)$ for computing the error correction. Note, that this error correction solver only generates relevant updates for the components located in the respective block.

We recall that in the implementation using block-asynchronous iteration as error correction solver, the asynchronism was limited to a stage where no component of the solution approximation could be updated a second time before all other components were updated and the new residual for the error correction solver was computed (see Algorithm 2). While the error correction solver itself was asynchronous, the synchronization between the iterative refinement steps did not allow for implementations leveraging the full (asynchronous)

Algorithm 22 Block-Asynchronous Iterative Refinement.

for all $(J_k \in \{J_1 \dots J_q\})$ **do** {asynchronous outer loop}
  read x from global memory
  compute residual $r_i = b_i - (Ax)_i$
  set $y = 0$
  for $(k = 0; k < iter; k++)$ **do** {equal local stopping criterion}
    for $(i = J_k(begin); i < J_k(end); i++)$ **do** {synchronous local updates}
      $y_i^{new} = K(y, r)$
      $y_i = y_i^{new}$
    end for
  end for
  update solution $x_i = x_i + y_i$
end for

computing power of hybrid hardware systems. Algorithm 22 avoids any synchronization steps, and therefore allows also for asynchronism in the iterative refinement process.

Compared to the block-asynchronous iteration directly applied to the linear problem the advantage is twofold: On the one hand, it enables the usage of different precision formats within the algorithm (see Algorithm 23), which may accelerate the overall process in case of a higher floating point performance in a precision format lower than working precision. A second advantage occurs due to the residual computation within the asynchronous process. Comparing the residuals we may obtain information about areas of faster and slower convergence. This would enable to adapt the number of iterations on the respective subdomains during runtime. In the end, the method should increase the local iteration count where necessary, and decrease it for components with small residuals. Note at this point, that this process is only for very diagonal dominant systems straight-forward. If the matrix is strongly coupled, the scheduling of the blockwise iterative refinement has strong impact on the method.

Algorithm 23 Block-Asynchronous Mixed Precision Iterative Refinement.

for all $(J_k \in \{J_1 \dots J_q\})$ **do** {asynchronous outer loop}
  read x^{high} from global memory
  compute error in high precision $r^{high} = b^{high} - A^{high}x^{high}$
  convert system to low precision for inner solver $A^{low} = A^{high}, \quad r^{low} = r^{high}$
  for all $l = 1 \rightarrow correction\_loops$ **do** {multiple error correction loops}
    set $c^{low} = 0$
    for $(k = 0; k < iter; k++)$ **do** {equal local stopping criterion}
      for $(i = J_k(begin); i < J_k(end); i++)$ **do** {synchronous local updates}
        $d_i^{low} = K(c^{low}, r^{low})$
        $c_i^{low} = d_i^{low}$
      end for
    end for
    convert error correction term to high precision $c^{high} = c^{low}$
    update solution $x^{high} = x^{high} + c^{high}$
  end for
end for

4.13.4. Experiments on Block-Asynchronous Iterative Refinement

In this section we analyze the convergence and performance characteristics of block-asynchronous iterative refinement in double and mixed precision mode. While the al-

gorithm layout is according to Algorithm 22, respectively Algorithm 23 for the mixed precision implementation, the block-asynchronous iterative refinement provides an even larger number of parameters that may be used for tuning the algorithm's performance. Among the block size and other values that could already be chosen in the block-asynchronous iteration we now can also set the number of relaxation steps of the error correction solver and the number of iterative refinement steps merged into one kernel. We desist from a deep analysis on how to optimize these values since the optimal choice may be very problem dependent. Particularly, very diagonal dominant problems may require less global communication than strongly coupled problems, and therefore allow a high iteration count for all loops on the distinct block-components. According to empirically based tuning we set for all experiments the number of relaxation steps to $iter = 15$ and the number of iterative refinement steps inside one kernel to $correction\_loops = 1$. Note however, that for diagonal dominant systems we may benefit from higher iteration counts.

The GPU implementations of the block-asynchronous iterative refinement methods were coded in CUDA [NVI09], whereas the respective libraries are taken from CUDA 4.0.17 [NVI11]. The kernels updating the respective components, launched through different streams, use thread blocks of size 512, equal to physical core number of the GTX580 GPU we target for these experiments. For hardware details see Table C.4 in the Appendix.

At this point we want to mention again, that also the results for the block-asynchronous iterative refinement have to be considered as average. While we desist from a detailed analysis like we do in Section 4.3 for the block-asynchronous iteration, we have to keep in mind that the convergence rate is influenced by the update order of the different blocks.

In a first experiment we analyze the convergence rate of the block-asynchronous iterative methods. Especially, we are interested in whether using single precision for the error correction solver has influence on the convergence rate. Figure 4.34 shows that the block-asynchronous iterative refinement converges for all test cases. Furthermore, using a lower precision format for the error correction computation has influence on the convergence of the overall method. While the variations between the block-asynchronous iterative refinement using double respectively single precision in the error correction solver are small for the diagonally dominant systems FV1 and FV3 (see Figure 4.34b and 4.34c), the convergence rate differs significantly for the matrices that contain larger off-diagonal parts (see Figure 4.34a, 4.34d). In the TREFETHEN_2000 case, the convergence rate is decreased by a factor of three when using single instead of double precision for the error correction part. For the system CHEM97ZTZ we not only have a lower convergence rate when using single precision error correction, but also observe non-consistent residual behavior. This may be an indication that the contraction property for the mixed precision iterative refinement is just barely fulfilled.

Again, the convergence rate alone is not sufficient to determine the algorithms' efficiency, as using low precision error correction may overcompensate the slower convergence by higher iteration performance. In Figure 4.35 we report the convergence with respect to time and observe that for the systems CHEM97ZTZ and TREFETHEN_2000 the higher single precision performance compensates for slower convergence (see Figure 4.35a, 4.35d). For the matrices where the convergence rate was almost not affected when replacing double precision error correction by the single precision variant, the overall runtime performance benefits significantly, see Figure 4.35b and 4.35c.

Probably the most interesting question is the trade-off between synchronous and asynchronous residual computation. For this reason we compare in Figure 4.36 the convergence of block-asynchronous mixed precision iterative refinement with the convergence of mixed precision iterative refinement using block-asynchronous iteration as error correction solver.

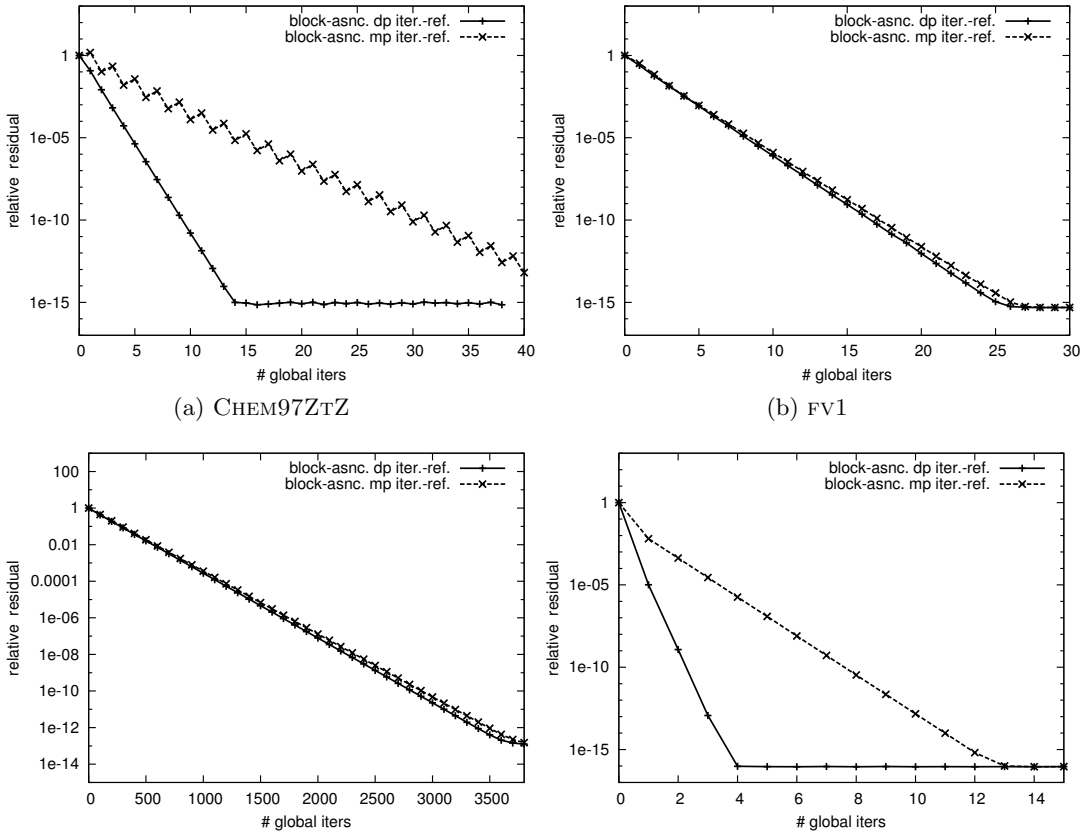
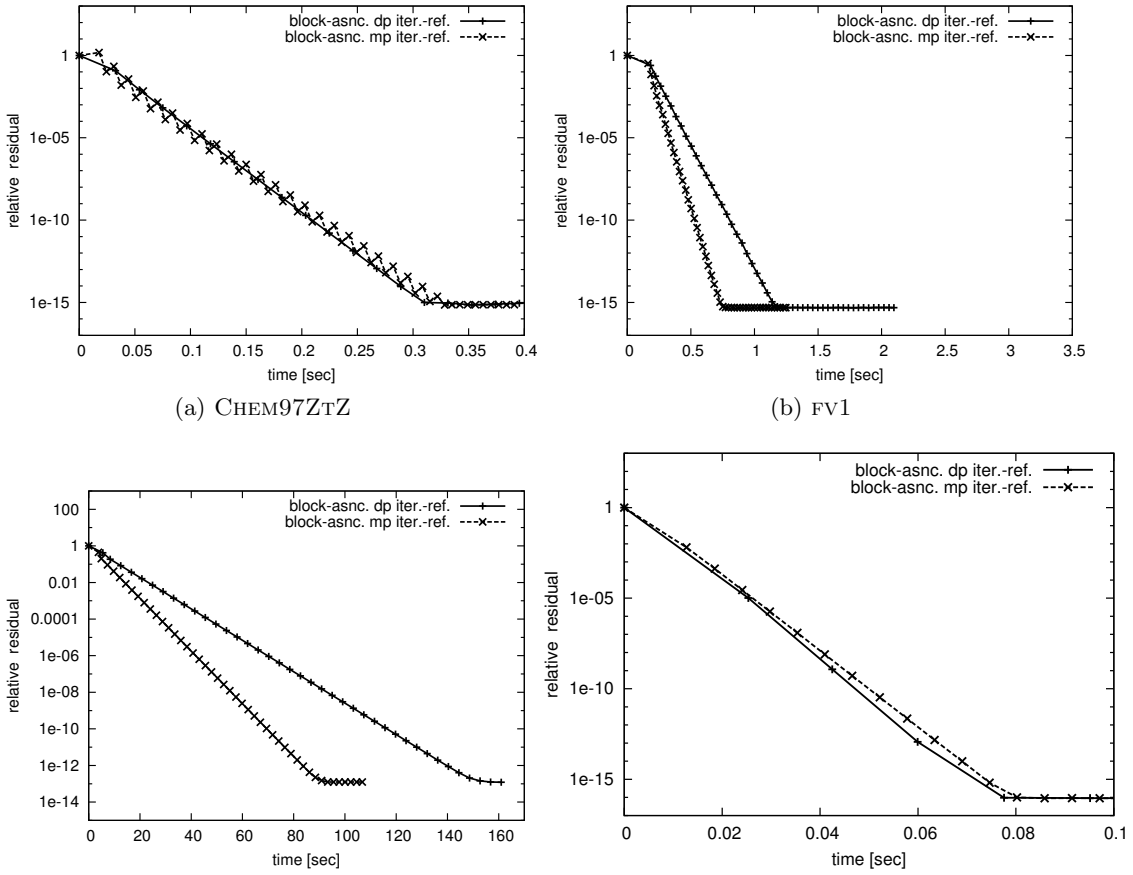


Figure 4.34.: Block-asynchronous iterative refinement convergence in double (dp) respectively mixed precision (mp) mode. The iteration-dependent relative residual is in L^2-norm.

These methods differ only in the residual computation, which is handled asynchronously or synchronously, respectively.

The results reveal a significantly slower convergence with respect to iteration numbers when handling the residual computation asynchronously. While the mixed precision iterative refinement using async-(5) as error correction solver never needs more than 5 iterations to converge, the block-asynchronous mixed precision iterative refinement needs at least three times as many outer iteration steps. But again, the omission of synchronization allows for a more efficient hardware usage, and the performance results from the trade-off between convergence rate and iteration rate. For this reason, we compare in Figure 4.37 the time needed by different block-asynchronous methods to provide a solution approximation of a certain relative residual accuracy. Note, that all solvers are based on block-asynchronous iteration, as the iterative refinement solvers in double (dp) and mixed precision mode (mp) use async-(5) to solve the error correction equations. Furthermore, despite the use of single precision in the mixed precision iterative refinement and the block-asynchronous mixed precision iterative refinement, all implementations provide a double precision solution approximation.

Analyzing the performance results we observe that general statements about the performance of the different methods are almost impossible, although using the same hardware configuration for all implementations. For the very diagonal dominant systems fv1 and fv3, replacing the synchronized iterative refinement methods with the block-asynchronous ones is not beneficial. Especially for the diagonal dominant system with low condition num-



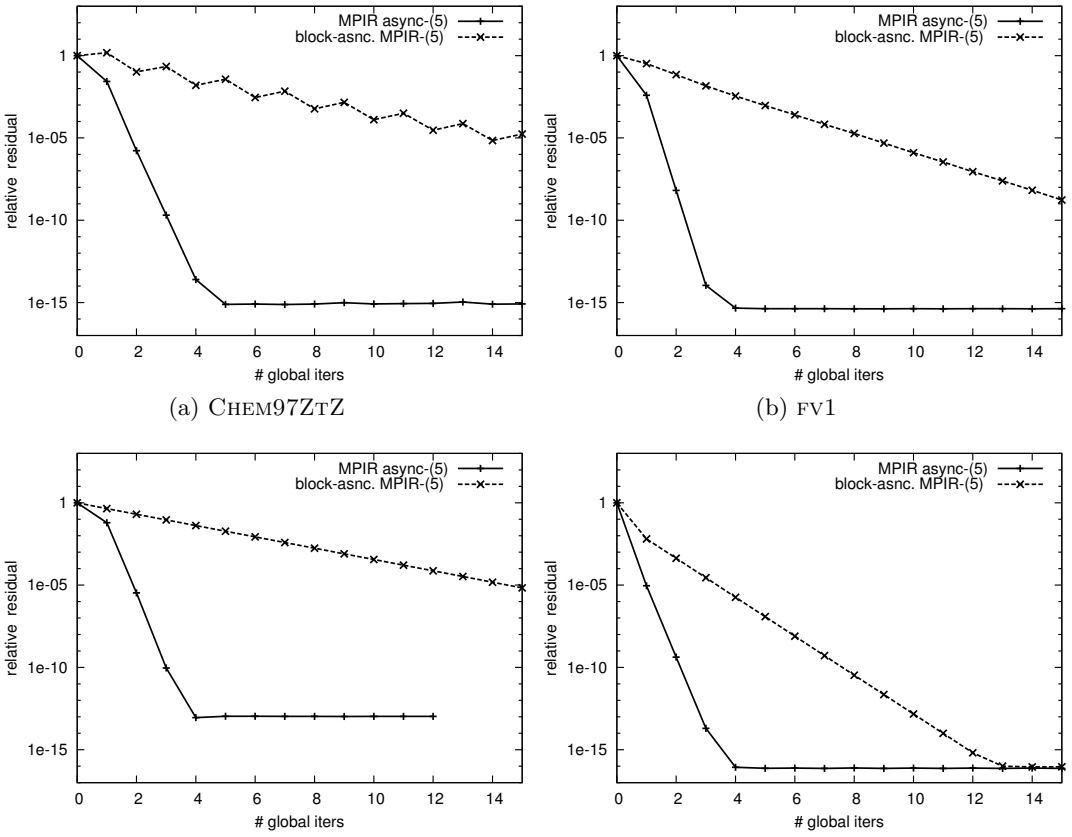
(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Figure 4.35.: Block-asynchronous iterative refinement performance in double (dp) respectively mixed precision (mp) mode. The relative residual is in L^2-norm.

ber (fv1), the runtime of the block-asynchronous iterative refinement exceeds the runtime of the synchronized one significantly. Using single precision for the error correction reduces the difference to the synchronized method, but the superiority of the synchronous variant remains. For the system fv3 with significantly higher condition number, the runtime differences between the synchronous and the asynchronous iterative refinement methods are smaller, both in double and mixed precision mode, see Figure 4.37c. Also, the performance in comparison to the plain async-(5) is improved: the block-asynchronous mixed precision iterative refinement outperforms the block-asynchronous iteration. When targeting the linear systems Chem97ZtZ and Trefethen_2000, both containing significant off-diagonal entries, the performance differences between the synchronized and block-asynchronous iterative refinement methods are small.

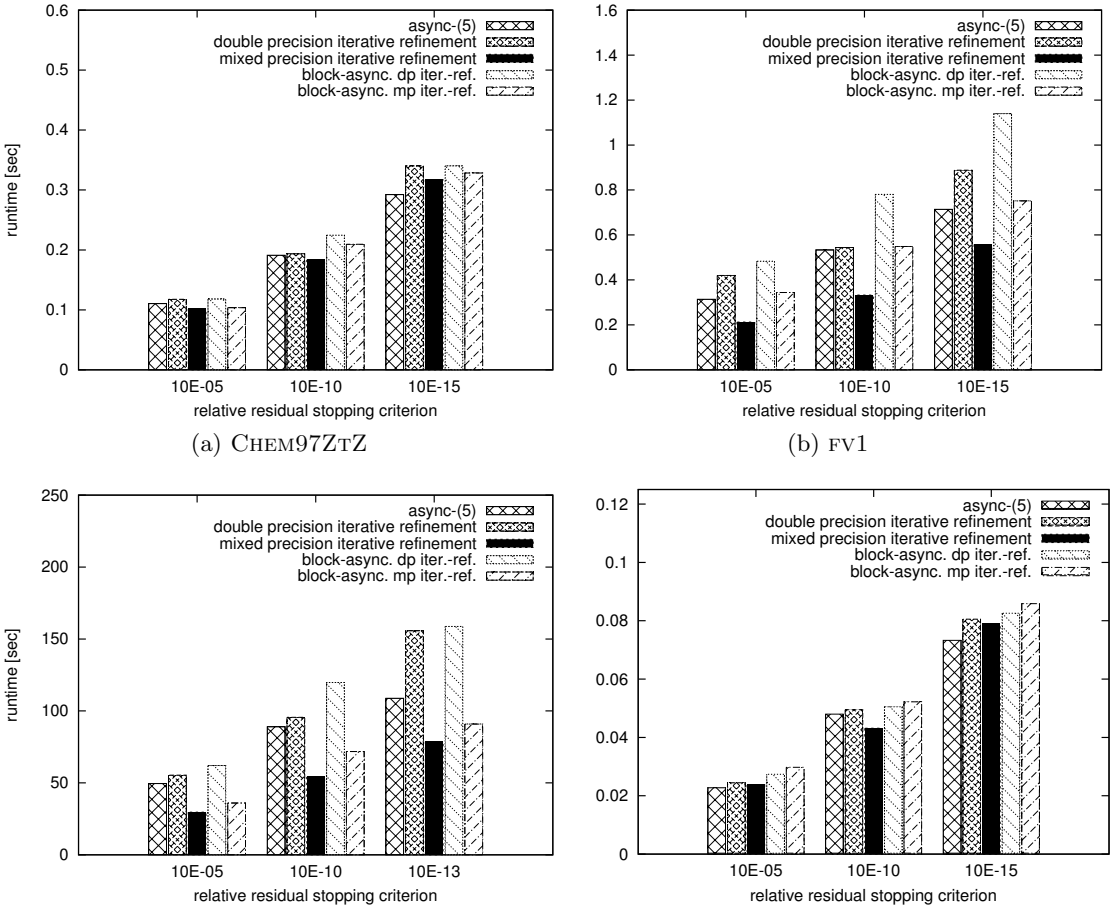
As a conclusion, we realize that the block-asynchronous iterative refinement did not outperform the synchronous iterative refinement methods for any test case. But at the same time we want to stress that the performance results are for one specific hardware configuration, different results may be possible when targeting other platforms. Especially the explosion in heterogeneity expected for future hardware systems may promote asynchronous methods. If the single precision performance remains higher than the double precision performance, the block-asynchronous mixed precision iterative refinement method may become interesting again.



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Figure 4.36.: Convergence comparison between mixed precision iterative refinement using async-(5) as error correction solver and block-asynchronous mixed precision iterative refinement. The relative residual is in L^2-norm.



(a) Chem97ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Figure 4.37.: Performance comparison between different block-asynchronous solvers. The relative residual stopping criteria are in L^2-norm.

4.14. Block-Asynchronous Iteration for Nonlinear PDEs

In the last sections we analyzed block-asynchronous iteration applied to linear systems of equations. Partial differential equations typically used for modeling physical, chemical or economical processes on the other side often also contain nonlinear parts. Since solving these nonlinear PDEs is an essential part of many computer simulations, much effort is put into deriving suitable algorithms. Solving nonlinear PDE's is computationally often costlier compared to linear problems, and the parallelization and optimization of nonlinear solvers is even more challenging. Against this background we want to address the question whether employing block-asynchronous iteration is beneficial when targeting the solution process of nonlinear PDEs.

One possibility to solve nonlinear equations is *Newton's*[2] *algorithm*, generating a sequence of linearized problems, that can then be handled by linear system solvers (e.g. block-asynchronous iteration). While every Newton step solves a linear problem, the sequence of solution approximations generated by the distinct Newton steps approaches (in the convergent case) a solution of the nonlinear problem. For a differentiable nonlinear operator $G : \mathbb{R}^n \to \mathbb{R}^n$ the Newton operator is given by [Ral79]

$$F(x) = x - \left(G'(x)\right)^{-1} G(x), \tag{4.17}$$

where $G'(x) = \left(\frac{\partial G_i}{\partial x_j}\right) \in \mathbb{R}^{n\times n}$ denotes the Jacobi matrix containing the partial derivatives of $G(x)$. This expression already implies that due to the required derivative of G the class of operators to which Newton's method can be applied is restricted by the assumption that G is differentiable. A comprehensive convergence theory on the Newton method is established by the *Kantorovich Theorem*[3] [HH07].

The idea for Newton's method can be derived from the *Taylor*[4] *approximation* of first order [SK06, QSS00]: To a given operator $G : \mathbb{R}^n \to \mathbb{R}^n$, the Taylor approximation of first order is given by

$$T_1(x, x^0) = G(x^0) + G'(x^0) \cdot (x - x^0).$$

Searching for a root x^1 to the linear problem $T_1(x, x^0) = 0$, we obtain

$$0 = G(x^0) + G'(x^0) \cdot (x^1 - x^0),$$

which brings us to the recursive Newton iteration operator

$$x^{k+1} = x^k - \left(G'(x^k)\right)^{-1} \cdot G(x^k).$$

To show the principles of the Newton linearization when solving a nonlinear PDE we consider the nonlinear system

$$-\Delta u = f(u) \quad u \in \Omega, \tag{4.18}$$

$$u = 0 \quad u \in \partial\Omega, \tag{4.19}$$

where $u : \Omega \to \mathbb{R}$ and $\Omega \subset \mathbb{R}^2$. This nonlinear partial differential equation is often used in computational chemistry to model difappliedfusion-driven chemical reactions [Gre65]. Also when analyzing diffusion-driven activator-inhibitor systems in Biology, e.g. pattern formation [Kot01], (4.18) is often the model of choice for the underlying processes.

[2] Sir Isaac Newton (⋆1642, †1727)
[3] Leonid Vitaliyevich Kantorovich (⋆1912, †1986)
[4] Brook Taylor (⋆1685, †1731)

Newton's method applied to this problem in continuous form we replace the nonlinear function $f(u)$ in (4.18) by its Taylor approximation of first order:

$$\begin{aligned} -\Delta u^{k+1} - \underbrace{f(u^{k+1})}_{\approx f(u^k)+f'(u^k)(u^{k+1}-u^k)} &= 0 \\ \rightsquigarrow \quad -u_{xx}^{k+1} - u_{yy}^{k+1} - f'(u^k)u^{k+1} &= f(u^k) - f'(u^k)u^k. \end{aligned} \tag{4.20}$$

We want to stress again that this linearization process requires the differentiability of the nonlinear function f [Ral79].

Once the problem is linearized, we can apply discretization techniques to obtain linear systems of equations for the distinct Newton steps. Assuming a unit square, a possible finite difference discretization can be obtained by the five-point stencil [Bra07].

Similar like in Section 4.10.7, for an equidistant mesh on the unitsquare with $N+2$ gridpoints in both directions, lexicographic numbering of the nodes [Bra07] and a node distance $h = \frac{1}{N+1}$, the sequence of discretized problems of size $n = N^2$ can then be written in the block system form

$$\begin{pmatrix} H & -I & 0 & \dots & 0 \\ -I & H & -I & & \vdots \\ 0 & -I & H & \ddots & \\ \vdots & & \ddots & \ddots & -I \\ 0 & \dots & & -I & H \end{pmatrix} \cdot u^{k+1} = b, \tag{4.21}$$

where

$$H = \begin{pmatrix} 4-h^2f'(u^k) & -1 & 0 & \dots \\ -1 & 4-h^2f'(u^k) & -1 & \\ 0 & -1 & 4-h^2f'(u^k) & \\ \vdots & & & \ddots \end{pmatrix} \in \mathbb{R}^{N\times N}, \tag{4.22}$$

$$I = \begin{pmatrix} 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \\ 0 & 0 & 1 & \\ \vdots & & & \ddots \end{pmatrix} \in \mathbb{R}^{N\times N}, \quad b = \begin{pmatrix} h^2\left(f(u^k) - f'(u^k)u^k\right) \\ h^2\left(f(u^k) - f'(u^k)u^k\right) \\ h^2\left(f(u^k) - f'(u^k)u^k\right) \\ \vdots \end{pmatrix} \in \mathbb{R}^n.$$

A possible implementation solving the nonlinear PDE by using Newton's linearization method and a discretization according to (4.21) is given in Algorithm 24.

4.14.1. Experiments on Block-Asynchronous Iteration applied to a Nonlinear Problem

As we have seen, Newton's method breaks down the solution process of the nonlinear problem into a sequence of linear problems, that can then be handled by linear equation solvers. When applying an iterative method in the distinct Newton steps, the question of optimal approximation accuracy arises. While a high accuracy approximation provides a good approximation for the linear problem solution, the high computational effort may not always be reasonable as the linear problem solution is only one contribution to the nonlinear solver. Using low accuracy approximations reduces the computation time in each Newton step, but usually increases the number of necessary Newton steps at the same time. In the end, the trade-off between iterations of the linear solvers in the Newton

Algorithm 24 Newton-linearization based block-asynchronous solver for nonlinear partial differential equations discretized by a five-point-stencil.

```
set u^0 = 0
for (k = 1; ; k++) do {iteration steps of Newton's method}
  assemble A and b according to (4.22)
  solve A u^{k+1} = b using block-asynchronous iteration
  if (|| u^{k+1} - u^k || <= ε || u^k ||) then {check error stopping criterion}
    break
  end if
  u^k = u^{k+1}
end for
solution approximation to problem: u^k
```

steps and the number of Newton steps applied to the nonlinear problem determines the overall solver performance. Note that this trade-off usually has to be considered for every problem individually.

For the nonlinear PDE problem (4.18), (4.19) with $f(u) = -(\alpha + \beta u^2)$ $(\alpha, \beta \in \mathbb{R})$, (4.20) becomes

$$-u_{xx}^{k+1} - u_{yy}^{k+1} + 2\beta u^n u^{k+1} = \beta \left(u^k\right)^2 - \alpha.$$

Using an equidistant 102×102 grid on $\Omega = [0,1] \times [0,1]$ we obtain for the fixed boundary conditions 100 unknowns per direction and a system dimension of 10000. Furthermore, we set $\alpha = 1$, $\beta = 2$ and analyse in the first experiment the trade-off between linear solver iterations and Newton steps when using Algorithm 24 with async-(5) (see Section 4.2). Additionally, we provide the convergence and performance of the Newton solver using the mixed-precision iterative refinement variant of async-(5) which we denote with mpir async-(5) (see Section 4.12), iterating until to a relative residual accuracy of $\frac{\|r^i\|_2}{\|r^0\|_2} \leq 10^{-10}$. As initial guess we set x^0 to zero.

The convergence results reported in Figure 4.38a are according to the expectations: A high accuracy approximation of the linear problem decreases the number of necessary Newton steps. But at the same time, the high iteration counts cause long runtimes for linear solver. In Figure 4.38b we observe, that the reduced number of Newton steps, coming at the cost of a more expensive linear solver, not necessarily accelerates the nonlinear solver. While leveraging the excellent low-precision performance in the mixed-precision iterative refinement variant maintains the superiority of the mpir async-(5) solver, the performance of the implementation using 1000 iterations of async-(5) is, despite the faster convergence in terms of Newton steps, inferior to the implementations applying less linear solver iterations. In the end, the reduced number of Newton steps does not pay off in terms of time-dependent convergence. For the remaining methods using 200, 250, 300, 350 or 500 iterations, the performance results are very similar. Concerning the approximation accuracy of the solution to the nonlinear problem we realize that high accuracy linear solvers are superior. But as only the last Newton step influences this approximation accuracy, an intelligent implementation would increase the iteration count of the linear solver with the number of Newton steps.

So far we have evaluated convergence and performance of block-asynchronous iteration in the context of Newton's method applied to a nonlinear problem. The essential question however is, whether it is a suitable replacement for the traditionally applied methods. In Figure 4.39 we compare the time-to-solution performance using different linear solvers in the Newton steps. While we use 500 iterations for the relaxation-based solvers (async-(5),

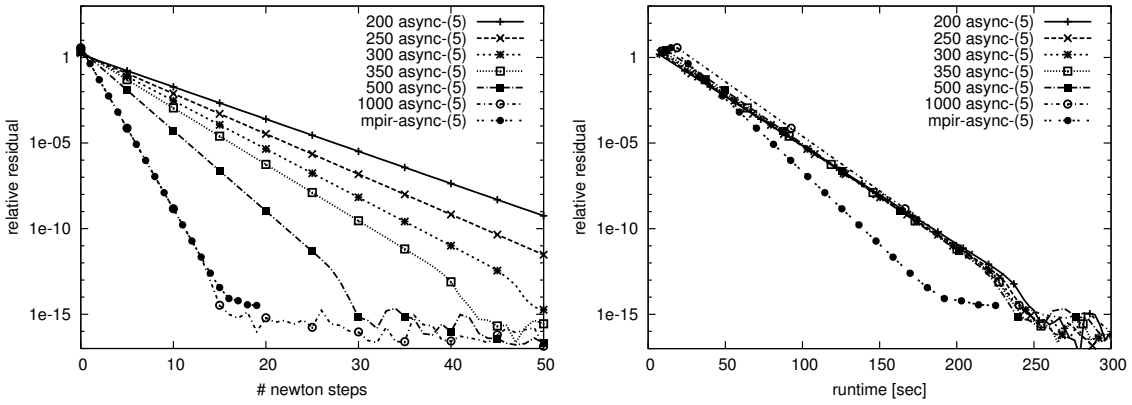


Figure 4.38.: Newton convergence using block-asynchronous iterative solvers for different iteration counts.

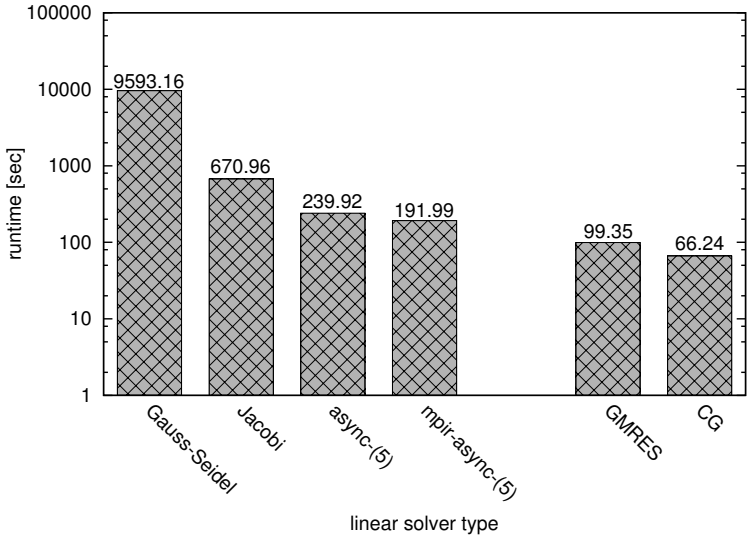


Figure 4.39.: Total runtime of Newton's method using different linear solver types applied to nonlinear problem. The relative residual stopping criterion of the Newton solver is set to $\frac{\|r^i\|_2}{\|r^0\|_2} \leq 10^{-15}$.

Jacobi, Gauss-Seidel) the Krylov-subspace methods iterate until convergence, which causes fast convergence of the Newton method in 3-5 steps. Note that only the Gauss-Seidel algorithm is CPU-based, all other solvers are implemented on the GPU. The superiority of the Krylov subspace solvers is expected, as they belong to the most efficient iterative methods when solving linear systems of equations. Compared to the other relaxation techniques, the block-asynchronous solvers are significantly faster. Comparing to the sequential Gauss-Seidel, the performance differs by almost two orders of magnitude, and in comparison with Jacobi, we still converge almost three times faster.

As we can expect an explosion in the heterogeneity of future hardware systems [BBC+08], the parameter computations requiring synchronizations in the Krylov subspace methods will become a crucial factor. Then, synchronization-avoiding algorithms will become the method of choice when solving nonlinear problems, which reveals the high potential of using block-asynchronous iteration also in the framework of nonlinear solvers.

4.14.2. Pattern Formation in Mathematical Biology

One of the still insufficiently answered questions in biology is which mechanisms are responsible for the typical coat pattern of the different species [Mur03]. The beautifully illustrated book series [Kin88] provides not only a comprehensive and accurate survey about the rich and varied spectrum of coat patterns, but also vividly shows that, despite the coat pattern of an individual mammal changes over lifetime, its species can always be identified. In mathematical biology, several models based on different mechanisms have been established that may provide an explanation for the pattern formation of animal coats. One popular model in this context is based on the idea, that one single process could be responsible for generating practically all of the common patterns observed. The model is based on a chemical reaction diffusion system, which is diffusionly driven, and the Turing[5] mechanism [Mur03]. The fundamental assumption that subsequent differentiation of the cells to produce melanin simply reflects the spatial pattern of morphogen concentration is compactly described by James D. Murray[6] [Mur03]: To create the color patterns certain genetically determined cells, called melanoblasts, migrate over the surface of the embryo and become specialized pigment cells, called melanocytes, which lie in the basal layer of the epidermis. Hair color comes from the melanocytes generating melanin, within the hair follicle, which then passes into the hair. The process of melanogenesis is comprehensively described by Giuseppe Prota in [Pro92]. As a result of graft experiments, it is generally agreed that whether or not a melanocyte produces melanin depends on the presence of a chemical, although we still do not yet know what it is. In this way the observed coat color pattern reflects an underlying chemical pre-pattern, to which the melanocytes are reacting to produce melanin [Mur03]. Without providing more details about the scale of the actual size of the pattern, the number of cells, the Turing instability range of the parameters and the motivation for the different mechanisms, we limit the background necessary to derive the mathematical model to the fact that two chemical substances with different diffusion characteristics (activator u and inhibitor v) are driving the chemical reaction diffusion process. The nonstationary partial differential equations describing this process are given by

$$\begin{aligned} \frac{\partial u}{\partial t} &= \gamma f(u,v) + \Delta u, \\ \frac{\partial v}{\partial t} &= \gamma g(u,v) + \kappa \Delta v \end{aligned} \tag{4.23}$$

where $f(u,v)$, $g(u,v)$ describe the interaction between the substances. Due to biological reasons it seems a natural decision to use Neumann[7] boundary conditions given by [Bra07]

$$\begin{aligned} \frac{\partial u}{\partial n} &= 0, \\ \frac{\partial v}{\partial n} &= 0. \end{aligned} \tag{4.24}$$

While there exist different reasonable choices for the functions $f(u,v)$ and $g(u,v)$, we stick to the Schnakenberg model [Sch79], which can be nondimensionalised and scaled into the general form [Mur03]

$$f(u,v) = \alpha - u + u^2 v, \tag{4.25}$$

$$g(u,v) = \beta - u^2 v. \tag{4.26}$$

[5] Alan Turing (⋆1912, †1954)
[6] James Dickson Murray (⋆1931)
[7] Carl Gottfried Neumann (⋆1832, †1925)

4.14.2.1. Discretization and Linearization

To solve this problem numerically by using an implicit Euler[8] scheme [But08] we choose a time discretization with timesteps dt. For the spatial discretization we assume a unit square and apply a finite difference discretization based on the five-point stencil with node distance h. We use the indices (i, j, k) where i, j refers to the spatial and k to the time discretization such that we get $x_i = i \cdot h$, $y_j = j \cdot h$ for $i, j = -1, 0 \dots n$ and $h = \frac{1}{n-1}$. Note that the additional ghost points $i, j = -1$ and $i, j = n$ are necessary for the discretization of the boundary with the symmetric difference quotient, and we have $x_0 = y_0 = 0$, $x_{n-1} = y_{n-1} = 1$, respectively. From (4.23) we derive

$$\frac{u_{i,j,k+1} - u_{i,j,k}}{dt} = \frac{u_{i+1,j,k+1} + u_{i-1,j,k+1} + u_{i,j+1,k+1} + u_{i,j-1,k+1} - 4u_{i,j,k+1}}{h^2} + \gamma f(u_{i,j,k+1}, v_{i,j,k+1}), \tag{4.27}$$

$$\frac{v_{i,j,k+1} - v_{i,j,k}}{dt} = \kappa \frac{v_{i+1,j,k+1} + v_{i-1,j,k+1} + v_{i,j+1,k+1} + v_{i,j-1,k+1} - 4v_{i,j,k+1}}{h^2} + \gamma g(u_{i,j,k+1}, v_{i,j,k+1}). \tag{4.28}$$

For a lexicographic ordering of the spatial discretization, we simplify the notation as follows: $U_k = u_{i,j,k}$, $V_k = v_{i,j,k} \in \mathbb{R}^{n^2}$, and $\bar{C} \in \mathbb{R}^{n^2 \times n^2}$ for the matrix referring to the Laplace operator with respective boundary conditions that were discretized by using a symmetric finite difference discretization. We then can write (4.27), (4.28) in a more compact fashion:

$$\frac{1}{dt}(U_{k+1} - U_k) = \frac{1}{h^2}\bar{C} \cdot U_{k+1} + \gamma \bar{f}(U_{k+1}, V_{k+1}), \tag{4.29}$$

$$\frac{1}{dt}(V_{k+1} - V_k) = \kappa \frac{1}{h^2}\bar{C} \cdot V_{k+1} + \gamma \bar{g}(U_{k+1}, V_{k+1}). \tag{4.30}$$

where

$$\bar{C} = \begin{pmatrix} C & 2I & 0 & \dots & 0 \\ I & C & I & & \vdots \\ 0 & I & C & \ddots & \\ \vdots & & \ddots & \ddots & I \\ 0 & \dots & & 2I & C \end{pmatrix} \in \mathbb{R}^{n^2 \times n^2} \text{with } C = \begin{pmatrix} -4 & 2 & 0 & \dots & 0 \\ 1 & -4 & 1 & & \vdots \\ 0 & 1 & -4 & \ddots & \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \dots & & 2 & -4 \end{pmatrix} \in \mathbb{R}^{n \times n},$$

$$\bar{f} : \mathbb{R}^{n^2} \times \mathbb{R}^{n^2} \mapsto \mathbb{R}^{n^2} \quad \bar{f}_{i,j} = f : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R},$$

$$\bar{g} : \mathbb{R}^{n^2} \times \mathbb{R}^{n^2} \mapsto \mathbb{R}^{n^2} \quad \bar{g}_{i,j} = g : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}.$$

Due to the nonlinear characteristic of the functions $f(u, v)$, $g(u, v)$, the implicit Euler method generates a sequence of nonlinear problems that we have to solve for the distinct timesteps. For this purpose, we apply the Newton linearization we already utilized in the last section, and employ block-asynchronous iteration to solve the linear systems.

Indexing the Newton iterations with l (the index for the time steps k remain constant in one time step) we obtain

$$F(U_{k+1}^l, V_{k+1}^l) + F'(U_{k+1}^l, V_{k+1}^l) \begin{pmatrix} U_{k+1}^{l+1} - U_{k+1}^l \\ V_{k+1}^{l+1} - V_{k+1}^l \end{pmatrix} = 0, \tag{4.31}$$

where $F : \mathbb{R}^{2n^2} \mapsto \mathbb{R}^{2n^2}$ with

$$F(U_{k+1}, V_{k+1}) = \begin{pmatrix} U_{k+1} - \frac{dt}{h^2}\bar{C} \cdot U_{k+1} - dt\gamma \bar{f}(U_{k+1}, V_{k+1}) - U_k \\ V_{k+1} - \kappa \frac{dt}{h^2}\bar{C} \cdot V_{k+1} - dt\gamma \bar{g}(U_{k+1}, V_{k+1}) - V_k \end{pmatrix}$$

[8] Leonhard Euler (⋆1707, †1783)

and $F' \in \mathbb{R}^{2n^2} \times \mathbb{R}^{2n^2}$ is the Jacobian with the partial derivatives with respect to U_{k+1} and V_{k+1}:

$$\begin{aligned}
&F'(U_{k+1}, V_{k+1}) \\
&= \begin{pmatrix} I - \frac{dt}{h^2}\bar{C} - dt\gamma \bar{f}_{U_{k+1}}(U_{k+1}, V_{k+1}) & -dt\gamma \bar{f}_{V_{k+1}}(U_{k+1}, V_{k+1}) \\ -dt\gamma \bar{g}_{U_{k+1}}(U_{k+1}, V_{k+1}) & I - \kappa\frac{dt}{h^2}\bar{C} - dt\gamma \bar{g}_{V_{k+1}}(U_{k+1}, V_{k+1}) \end{pmatrix} \\
&= \begin{pmatrix} I - \frac{dt}{h^2}\bar{C} - dt\gamma \cdot (-I + \mathrm{diag}(2U_{k+1}. * V_{k+1})) & -dt\gamma \cdot \mathrm{diag}(U_{k+1}. * U_{k+1}) \\ -dt\gamma \cdot \mathrm{diag}(-2U_{k+1}. * V_{k+1}) & I - \kappa\frac{dt}{h^2}\bar{C} - dt\gamma \cdot \mathrm{diag}(-U_{k+1}. * U_{k+1}) \end{pmatrix}.
\end{aligned}$$

Note that we use the Matlab[9]-notation $X. * Y$ for the component wise multiplication of two vectors X, Y. Using

$$\begin{aligned}
&F'(U_{k+1}^l, V_{k+1}^l) \begin{pmatrix} U_{k+1}^{l+1} - U_{k+1}^l \\ V_{k+1}^{l+1} - V_{k+1}^l \end{pmatrix} \\
&= F'(U_{k+1}^l, V_{k+1}^l) \begin{pmatrix} U_{k+1}^{l+1} \\ V_{k+1}^{l+1} \end{pmatrix} - F'(U_{k+1}^l, V_{k+1}^l) \begin{pmatrix} U_{k+1}^l \\ V_{k+1}^l \end{pmatrix}
\end{aligned}$$

we can derive the sequence of linear systems of equations that have to be solved in the distinct Newton steps l:

$$F'(U_{k+1}^l, V_{k+1}^l) \begin{pmatrix} U_{k+1}^{l+1} \\ V_{k+1}^{l+1} \end{pmatrix} = F'(U_{k+1}^l, V_{k+1}^l) \begin{pmatrix} U_{k+1}^l \\ V_{k+1}^l \end{pmatrix} - F(U_{k+1}^l, V_{k+1}^l), \tag{4.32}$$

where

$$\begin{aligned}
&F'(U_{k+1}^l, V_{k+1}^l) \begin{pmatrix} U_{k+1}^l \\ V_{k+1}^l \end{pmatrix} \\
&= \begin{pmatrix} U_{k+1}^l - \frac{dt}{h^2}\bar{C}U_{k+1}^l - dt\gamma(-U_{k+1}^l + \mathrm{diag}(2U_{k+1}^l. * V_{k+1}^l)U_{k+1}^l) - dt\gamma \cdot \mathrm{diag}(U_{k+1}^l. * U_{k+1}^l)V_{k+1}^l \\ -dt\gamma \cdot \mathrm{diag}((-2U_{k+1}^l. * V_{k+1}^l))U_{k+1}^l + V_{k+1}^l - \kappa\frac{dt}{h^2}\bar{C}V_{k+1}^l - dt\gamma \cdot \mathrm{diag}(-U_{k+1}^l. * U_{k+1}^l)V_{k+1}^l \end{pmatrix}.
\end{aligned}$$

In order to better map the system to the number range of computers, we multiply the obtained equation by h^2 and obtain a linear system $Ax = b$ with

$$b = \begin{pmatrix} h^2 dt\gamma \left(\alpha \cdot I - 2 \cdot \mathrm{diag}(U_{k+1}. * U_{k+1})V_{k+1}^l\right) + h^2 U_k \\ h^2 dt\gamma \left(\beta \cdot I + 2 \cdot \mathrm{diag}(U_{k+1}. * U_{k+1})V_{k+1}^l\right) + h^2 V_k \end{pmatrix} \tag{4.33}$$

and

$$A = \begin{pmatrix} A_u & R \\ Q & A_v \end{pmatrix}, \tag{4.34}$$

where for $t \in \{u, v\}$

$$A_t = \begin{pmatrix} H_t & -2Idt & 0 & \dots & 0 \\ -Idt & H_t & -Idt & & \vdots \\ 0 & -Idt & H_t & \ddots & \\ \vdots & & \ddots & \ddots & -Idt \\ 0 & \dots & & -2Idt & H_t \end{pmatrix} + h^2 q_t$$

[9] MathWorks; `www.mathworks.com/`

and

$$\begin{aligned} H_u &= h^2 I - dtC \\ q_u &= dt\gamma(I - 2 \cdot \mathrm{diag}(U^l_{k+1}. * V^l_{k+1})), \\ H_v &= h^2 I - \kappa dtC \\ q_v &= dt\gamma \cdot \mathrm{diag}(U^l_{k+1}. * U^l_{k+1}), \\ R &= -h^2 dt\gamma \cdot \mathrm{diag}(U^l_{k+1}. * U^l_{k+1}), \\ Q &= h^2 dt\gamma 2 \cdot \mathrm{diag}(U^l_{k+1}. * V^l_{k+1}). \end{aligned}$$

Notice again, that the matrix C referring to the Laplace operator comprises the boundary condition (4.24) using centralized symmetric finite differences that allow an accuracy of $\mathcal{O}(h^2)$.

Since we want to solve the linearized system by applying the block-asynchronous iteration, we recall that the method not necessarily converges for $\rho(|B|) > 1$ where B is the iteration matrix $B = I - D^{-1}A$, while a sufficient condition for convergence is $\rho(|B|) < 1$ (see Theorem 3.2.6). Analyzing the structure of the system matrix A, we notice that by choosing the time discretization dt small enough, we are able to ensure $\rho(|B|) < 1$. Then, the sufficient convergence condition for block-asynchronous iteration is fulfilled, which shows that the method can efficiently be applied also when solving nonlinear nonstationary problems.

4.14.2.2. Simulations using Block-Asynchronous Iteration

When using time-discretization methods with variable time steps, the comparison between different solver algorithms is in general difficult, as we have to choose the time discretization dt small enough such that the corresponding iteration matrix fulfills $\rho(|B|) < 1$ and the block-asynchronous iteration converges. Hence, we have less freedom in the choice of the time step length than when applying other methods to solve the linear systems. This implies that for other methods, like e.g. Gauss-Seidel or GMRES (see Section 2.4.2, respectively 2.5.4), it may be possible to choose a coarser time discretization, but at the same time, these methods allow only limited parallelization and require synchronization between the iterations, which may become crucial when targeting highly parallel heterogeneous hardware.

The choice of parameters and initial solution approximation is according to the experiment setup chosen in [Ruu95] (also see [FF12, Mad06]):

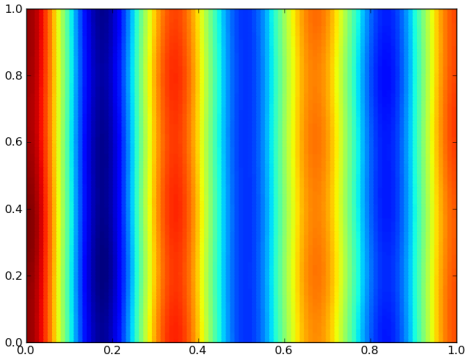
$$\begin{aligned} \Omega &= [0,1] \times [0,1], \\ \kappa &= 10.0, \\ \gamma &= 1000.0, \\ a &= 0.126779, \\ b &= 0.792366, \\ \kappa &= 10.0, \end{aligned}$$

$$U_0 = 0.919145 + 0.0016 \cdot \cos(2\pi(x+y)) + 0.001 \sum_{j=1}^{8} \cos(2\pi jx),$$

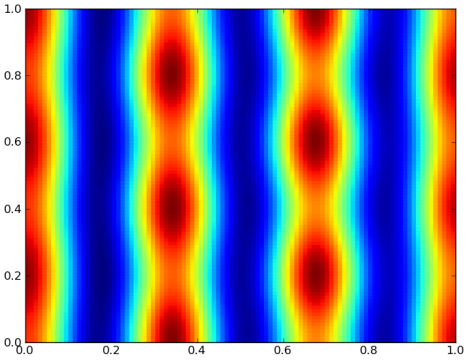
$$V_0 = 0.937903 + 0.0016 \cdot \cos(2\pi(x+y)) + 0.001 \sum_{j=1}^{8} \cos(2\pi jx).$$

Only, since we are able to exploit the parallelism of hybrid hardware systems, we choose a discretization of 100 elements per direction, resulting in a total number of 20.000 unknowns, and a time discretization $dt = 0.1 \cdot h^2$.

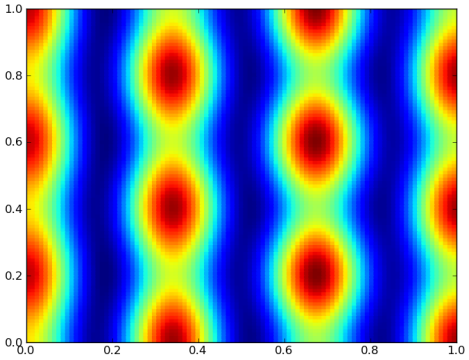
In Figure 4.40 the distribution of the activator u is visualized for various values of t. We observe, that the system converges towards a steady pattern distribution, which may for this showcase result in a dotted coat. A comparison with simulation results obtained by using a direct solver for the linearized problems reveals negligible differences to the obtained solution approximations (also see simulation results in [FF12]).



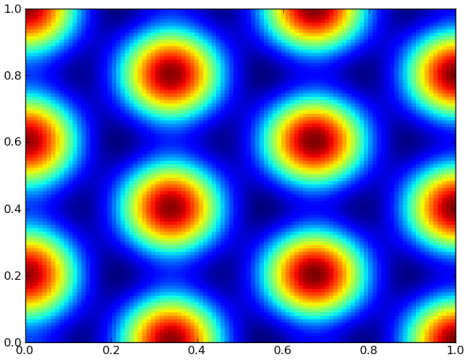
(a) t=0.025



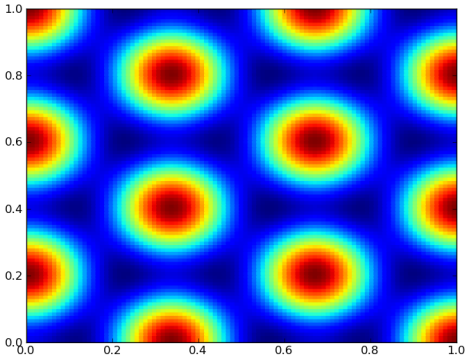
(b) t=0.125



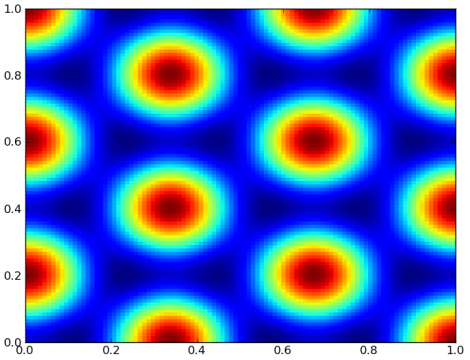
(c) t=0.15



(d) t=0.2



(e) t=0.225



(f) t=5.0

Figure 4.40.: Visualization of the distribution of activator u for different values of t.

5. Power-Aware Implementations and Energy-Efficient Numerics

5.1. Power- and Energy-Efficiency

High Performance Computing (HPC) centers are substantial consumers of energy, necessary to feed the computational resources that have enabled the breakthrough scientific advances achieved during the past few decades. The recent developments in computer architecture, especially in multi-/many-core and accelerator technology, have triggered considerable performance gains in computing, allowing the continuation of historical trends [top], and this hardware is being rapidly adopted in HPC facilities. Nevertheless, further performance improvements attained from a substantial increase in the number of cores, is constrained by the aggregated energy budget necessary for large-scale HPC systems. In particular, power consumption has a direct impact on the operational costs of these centers, compromising their existence and impairing the installation of new ones. Already today, the electricity costs for many HPC centers exceed the hardware acquisition costs in just few years [Lam10]. Also the economic issue is not the only problem of this immense hunger for energy. Appropriate infrastructure able to supply this enormous amount of energy is not always available, and also cooling the facilities may become a serious problem, especially since the heat dissipation reduces the reliability and lifetime of hardware components [NX06, SSH+03].

The needed amount of energy has to be generated somehow, and considering the current energy mix, also the ecological impact has to be considered, as the associated carbon dioxide emission is a hazard for the environment and public health. Therefore, the concerns about the rise of an energy crisis, climate change and fault-tolerance in large-scale systems lead to a very well justified call for energy efficiency in HPC [BBC+08, D*et al*11]. One consequence is the establishment of the Green500 list [gre], ranking the supercomputers according to their performance per watt ratio, as counterpart to the performance-oriented TOP500 list [top]. The fact that at the time of writing in June 2012 for the first time the No. 1 supercomputer in the Green500 list was based on the same architecture like the system ranked number one in the TOP500 list revels the increasing relevance of the power constraint and the strong efforts of the hardware developers to develop low-consuming systems [top, gre]. In many cases, the gain in power efficiency is achieved by integrating low-consuming accelerators in the platform that provide a high performance per watt ratio [BTL10]. A strong correlation between the integration of power-efficient coprocessors

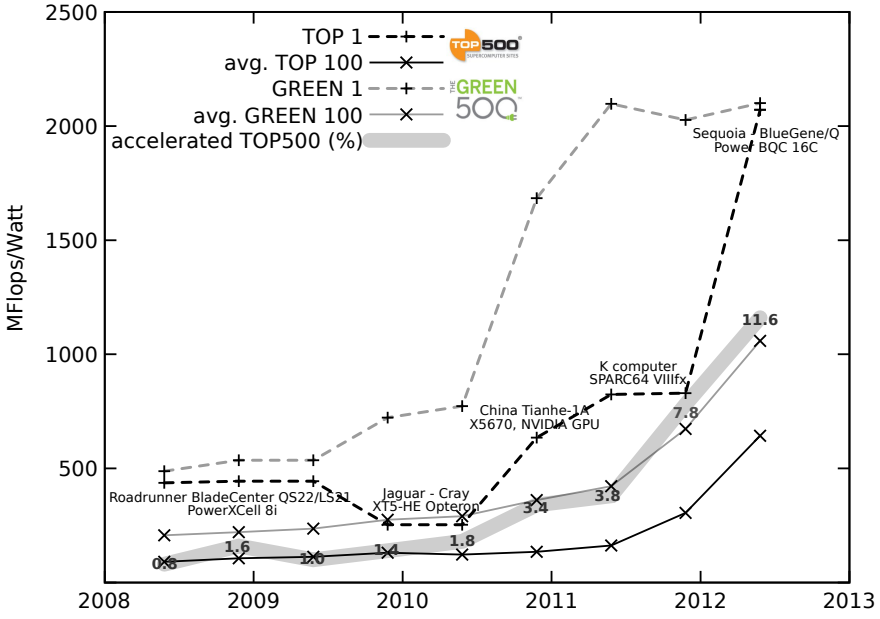


Figure 5.1.: Energy-dependent performance of the top-ranked systems in the TOP500 and GREEN500 lists [top, gre]. The dashed lines are for the No. 1 ranked system, the solid lines the average of the first 100 systems in the list. The wide line gives the percentage of the TOP500 systems featuring accelerators.

and a higher energy efficiency can also be assumed by comparing the percentage of accelerated supercomputers and the energy efficiency of the top-ranked systems in Figure 5.1.

Despite the recent improvements, all technical reports identify the power consumption of the computers as the largest hardware research challenge when aiming for exascale computing [BBC+08, Age10, GL09]. The fact that using today's hardware would result in power draft of over one gigawatt for exascale systems shows the need for further efforts of the hardware developers [ABC+10].

At the other end of the spectrum, mobile computing devices need to be equipped with low-power hardware components to maximize their battery life. This constraint particularly from the embedded and mobile market segments, is forcing hardware manufacturers to improve their designs for better energy efficiency. Processor, memory and hard disks nowadays feature low-power modes to trade-off performance for power by applying energy-friendly techniques such as Dynamic Voltage and Frequency Scaling (DVFS, see Section 5.3.2) and idle states (e.g., spin down idle disk platters). These tools have to be used carefully, though, as the application runtime increase may outweigh the power decrease such that the total energy is increased. In recent years, these energy-saving mechanisms from the mobile market have found their way into server architectures and, thus, the systems installed at large HPC centers are now often equipped with power-efficient hardware. However, the system software, communication libraries, and application codes in these systems are most often insensitive to power consumption.

Summarizing the different aspects we may predict, that, although the exaflop challenge will undoubtedly lead to new ground-breaking scientific discoveries, it is also certain that it calls for greener and more efficient resource utilization. This includes not only low-consuming hardware forming the HPC facilities, but also software, middleware and application algorithms that are able to leverage the energy saving techniques.

It is clear that exascale machines will not be materialized unless the "power wall" problem

is adequately addressed [BBC+08]. A non-exhaustive list of critical factors with respect to energy reduction encompasses: facility, hardware, software, scheduling, and energy management. It is well known that a large potential for energy reduction can be obtained by means of re-engineering facility and hardware. This topic is already fully included in the focus of computer manufacturers (see e.g. [YHE02], [BB95], [GC01], [Hof05], [KGMS97]).

On the other side, little attention is payed to the fact that a great potential can be leveraged by transforming, redesigning and completely rethinking the algorithms that are implemented in the applications that run on the supercomputing platforms. The difficulty, and possibly one of the reasons why only few research institutions investigate this topic in a systematic way, is related to the fact that this kind of redesign strategy relies on an highly interdisciplinary expertise coupling the disciplines of mathematical modeling, numerical methods, software design and hardware aware computing.

While the different experts can improve the factors related to the respective fields they are working in, only the combination of their competences can lead to considerable improvements: the hardware has to be optimized with respect to power consumption, the applications have to be adapted to leverage this hardware, the implementations have to optimize the usage of all available hardware resources and distribute the workload to improve the efficiency.

In the following sections we want to investigate experimentally how the iterative methods we derived and analyzed in the last chapters can be adapted and optimized with respect to the power and energy dissipation, see Section 5.4. This includes the issue of the efficient utilization of power saving mechanisms provided by the hardware. To understand these, we provide in Section 5.3 a broad overview about the different concepts. As accurate power quantifications are substantial for the analysis, we start in Section 5.2 with a description of the measurement setups for the targeted systems.

5.2. Measurement Setup for Energy Analysis

Before reporting on the experiments and the respective results, we want to provide some information about how to monitor the energy footprint of a scientific simulation algorithm conducted on a specific hardware without affecting the system itself. The idea is to use an independent measurement setup, reporting the voltage and amperage of the different devices. Two factors limit the accuracy of these measurements: The limited frequency of the analyzing instruments and the damping devices integrated in modern hardware. These damping devices, usually integrated capacitors, aim for buffering current and amperage peaks, and hence lead to a smoother power draft. But while this potentially improves the power consumption of the applications, at the same time it aggravates the analysis of the power profile of the respective application. Hence, it is reasonable to profile computationally expensive problems, as we may then expect long routine calls and the amperage to settle down at the correct value.

5.2.1. Power Measurement Setup at the University of Jaume I

The *High Performance Computing & Architectures (HPCA)*[1] group at the University of Jaume I, Castellon, has employed a power analysis setup able to monitor the power demand of the mainboard and the GPU of a hybrid hardware system separately. The hardware system of Watts-2 (see Appendix C.2) processing the scientific applications consists of two Intel Xeon E5410 Quad-Core processors at 2.33 GHz, with 4 GB of RAM, connected via PCIe (16x) to a GPU. Power is measured at two different points, Figure 5.2 illustrates the

[1] High Performance Computing & Architectures (HPCA); `http://www.hpca.uji.es/`

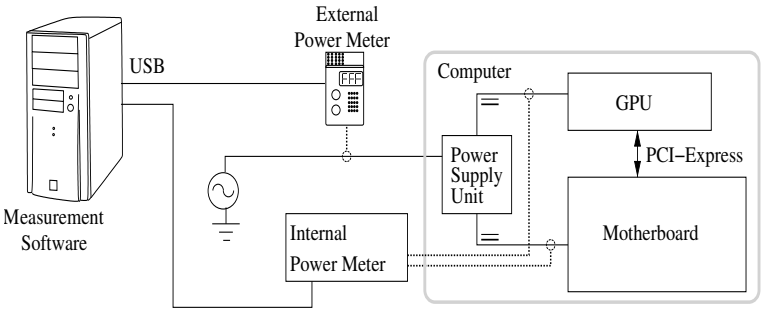


Figure 5.2.: Hardware platform and sampling points at the University of Jaume.

connection between the target platform and the energy measurement hardware [AHR+11a]. A commercial external power meter samples power for the global system once per second (1 Hz). Given the low resolution of the measurements, and the "noise" introduced by hardware components as the disk or the network interface card as well as the inefficiencies of the power supply, there is an alternative sampling point added, with a higher resolution, using an internal power meter. This is an ASIC operating at a frequency of 25 Hz (25 samples per second) which is composed of a number of resistors connected in series with the power source: thus, the drop of power voltage across the series yields a direct estimation of the power being consumed. The internal powermeter is attached to the lines connecting directly the power supply unit with the GPU and the motherboard (chipset plus processors) so as to obtain the energy consumption of the computing hardware. Since the samples from both the external and the internal power meter are collected in a separate measurements system, the measurements do not affect the performance of the application.

5.2.2. Power Measurement Setup at Engineering Mathematics and Computing Lab (EMCL)

Inspired by the installation at the University of Jaume we decided to set up a similar system at the Engineering Mathematics and Computing Lab at the Karlsruhe Institute of Technology[2]. The objective was to integrate the Supermicro GPU Cluster (see Appendix C.1) into a measurement setup able to monitor power and amperage of the distinct hardware devices. Funded by the Karlsruhe Institute of Technology we started a cooperation project with the Institute for Data Processing and Electronics (IPE)[3] located at the north campus of the Karlsruhe Institute of Technology. With the help of Dr. Andreas Kopmann, Armen Beglarian and Peter Schöck we configured a system that is not only able to monitor voltage and amperage of the chipset including the memory, the processors and the interface, but also the hard disks, the GPUs, and even the power draft of the GPUs via the PCI 16x connection. While Figure 5.3 provides a general overview, which may be considered as a worldwide unique system setup, more details about the configuration of the measurement system can be found in Appendix C.1.

5.3. Energy Saving Techniques

Over the last years, hardware developers have devoted strong efforts to the design of power-efficient devices. As one result, parallel coprocessors like GPUs, FPGAs and Manycore-systems like Intel's MIC were promoted, that provide excellent floating point performance with low power dissipation. Additionally, modern hardware often features mechanisms to reduce the power demand, but the crucial question has become how to leverage these

[2]Engineering Mathematics and Computing Lab (EMCL); `http://www.emcl.kit.edu/`
[3]Institute for Data Processing and Electronics (IPE); `http://www.ipe.kit.edu/`

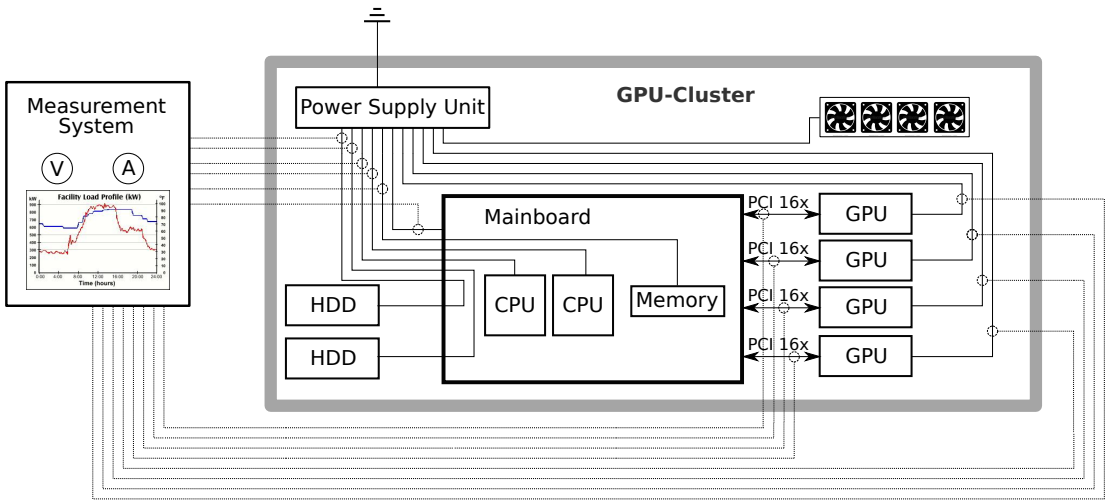


Figure 5.3.: Hardware platform and sampling points at the EMCL.

from the software implementations. Dynamic Voltage and Frequency Scaling (DVFS) and idle-wait, reviewed next, are two simple techniques that can efficiently be embedded into an algorithm, to exploit hardware-provided power saving mechanisms, and to reduce the power dissipation and energy consumption of an application.

5.3.1. Energy Improvements by Accelerator Technology

Using Accelerator technology for applications first of all implies an additional power consumer. Every additional device not only needs additional power, but also the associated memory, cooling and the interface to the coprocessor raise the overall dissipation. Nevertheless, using accelerator technology may still be beneficial to the total energy consumption of the application since the high computational power often improves the performance. In this case, the higher average power demand may be compensated for by a more decreased overall runtime [ARH10]. As it depends on the system architecture and the application whether adding coprocessors improves the energy footprint, we will analyze in Section 5.4.1 the situation for iterative linear system solvers on specific hardware configurations applied to some test problems.
At this point we want to stress that the total energy consumption alone may not always be the relevant metric, since in many cases additional constraints are posed by the infrastructure and other factors. As an example, for many systems the peak power, including the demand for power supply and cooling system, poses restrictions to the configuration, as the overall power throughput may be limited by the infrastructure. Related to this issue is the need for suitable energy metrics that reflect the power and energy profile of an application: Focusing on the total energy consumption ignores boundary conditions like limited CPU (or coprocessor) hours, the power dissipation of the infrastructure or the simulation's relevance at a certain point in time [BC10].

5.3.2. Dynamic Voltage and Frequency Scaling (DVFS)

The power demand of a computing processor depends linearly on its capacitance and its clock speed, and quadratically on its voltage. The dependencies can be formulated in a general model for static CMOS gates as

$$P = C \cdot U^2 \cdot f, \tag{5.1}$$

where C is the capacitance being switched per clock cycle, U is the voltage and f is the switching frequency [RCN04]. Although (5.1) is a simplification (most modern processors

are not plain CMOS architectures, and the formula neglects static leakage), the expression captures a key point, stating that the power draft depends linearly on the clock speed and quadratically on the voltage. Hence, the processor frequency and the voltage are two parameters that can be adjusted to reduce the overall power draft. The problem is that they are related and, therefore, changing one of them usually requires the adaption of the second, so that in general frequency and voltage have to be scaled simultaneously. This can be derived from the fact, that reducing the voltage requires a slower clock rate to fill the capacitors in the processor.

Dynamic Voltage and Frequency Scaling (DVFS, see [CSP04, SMB+02]) is an efficient technique that aims at reducing the power dissipation by applying the idea of the simultaneous rescaling of voltage and frequency. Specifically, note that a linear reduction of the voltage and the clock speed in the same proportion potentially triggers a cubic reduction of power, see (5.1). The drawback is that reducing frequency by a given factor usually comes at the cost of a negative performance impact, so that the application runtime is usually increased. As the overall energy need is the product of the computation time and power draft, the result may be a growing total energy consumption [FLP+07, pow09]. Particularly, for CPU-intensive codes, applying DVFS is a trade-off between extending the overall computation time and lowering the power draft. While reducing voltage and frequency necessarily has a positive impact on the processor's energy demand, this is not always true when considering the complete hardware system. The reason is that also the memory, the hard disk, the network and other hardware components contribute to the total energy consumption and, thus, the extended runtime may blur energy savings achieved for the CPU. Nevertheless, especially for small devices like smart-phones, tablets and embedded systems, where the processor is the dominating power consumer, DVFS has been largely adopted as an efficient energy-saving mechanism.

5.3.3. Idle-Wait

When conducting operations on a coprocessor, e.g. running a kernel routine on a GPU, the CPU of the host system is either used for other tasks or runs idle. Since many algorithms do not allow sharing CPU resources with some other application, and also demand for the completion of the coprocessor task before continuing the program execution, it is reasonable to reduce the processor power for the time of the kernel execution. One possibility to achieve this goal is to simply apply DVFS to the CPU for the duration of the kernel execution. Nevertheless, in experiments it was observed, that that lowering the operation frequency of idle CPU cores rarely yields a significant reduction of power consumption of the host system [AHA+11]. The reason for this is the "busy-wait" status the CPU enters for the time of the kernel execution, which is highly energy inefficient. To resolve this problem, in [AHA+11] an alternative "idle-wait" technique was proposed, which is able to reduce the CPU energy consumption for the time of the kernel call considerably (also see [ACF+]). The underlying idea is to explicitly set the host system to sleep for approximately the time of the kernel execution. In the past only possible by estimating the kernel time a priori and utilizing the C/C++ `nanosleep()` function (see `sys/time.h`) [AHA+11], meanwhile the new CUDA toolkit provides tools that allow a very flexible handling of idle-wait [ADI+12]. In particular, `cudaSetDeviceFlags` allows to specify the behavior of the active host thread when it executes device kernel code. This routine is called by the CPU thread before the CUDA runtime is initialized. After that, all synchronizations using `cudaThreadSynchronize` will suspend the execution of the calling thread until the device finalizes its work, thus "sleeping" the core and avoiding the potential energy-consuming state. Note that applying idle-wait to any coprocessor-accelerated application necessarily improves its energy consumption without impacting its performance. This stems from the fact that it only adapts the CPU power when running idle, and the algorithm's runtime

performance is not affected by the mechanism.

5.4. Experiments on Energy Saving Techniques

5.4.1. Experiments on GPU-Accelerated Numerics

In the first experiment we aim for analyzing the tradeoff between the reduced runtime of GPU-accelerated code and the additional power demand of the coprocessor. The results reported in this and the two following sections were also presented in [AHA+11]. On the watts-2 system (see Appendix C.2) located at the University of Jaume I we perform a detailed power and energy analysis on a CG solver and the respective Jacobi-preconditioned variant applied to the system of linear equations based on the matrix G3_CIRCUIT (see Appendix B). We differentiate between CPU implementations using 1,2,4 and 8 cores, and a GPU implementation outsourcing the matrix vector operations to a Tesla C1060 board (see Appendix C.4).

Note that since the computation of the preconditioner matrix is sequential, it is always performed by the CPU of the system. Within the iteration process, the difference between the CG and the Jacobi-preconditioned CG (PCG) is one additional matrix-vector multiplication involving the preconditioner matrix.

The different implementations of the solver use either the CPU or the GPU for the sparse matrix-vector product and BLAS-1 operations.

In Tables 5.1 and 5.2 we report the results obtained with the CPU solvers using OpenMP with 1, 2, 4 and 8 threads/cores (results labelled as "CPU nT" with "n" equal the number of threads), and compare the execution time (in seconds, [sec]) and the energy consumption (in Watts-hour, [Wh]) to those of the GPU implementations, performing the BLAS-1 operations and the sparse matrix-vector product on the GPU, and using a total of 4 CPU threads/cores for the remaining operations on the CPU (results labelled as "GPU 4T").

hardware	# iter	time [sec]	energy consumption [Wh]		
			chipset	GPU	total
CPU 1T	21424	1674.45	53.96	-	53.96
CPU 2T	21424	1307.21	45.70	-	45.70
CPU 4T	21424	1076.97	42.18	-	42.18
CPU 8T	21424	1113.34	50.54	-	50.54
GPU 4T	21467	198.43	8.04	3.44	11.48

Table 5.1.: Energy consumption of different implementations of CG solver for G3_CIRCUIT [AHA+11].

hardware	# iter	time [sec]	energy consumption [Wh]		
			chipset	GPU	total
CPU 1T	4613	601.97	18.94	-	18.94
CPU 2T	4613	417.33	14.22	-	14.22
CPU 4T	4613	348.79	13.31	-	13.31
CPU 8T	4613	362.44	16.25	-	16.25
GPU 4T	4613	46.28	1.89	0.83	2.72

Table 5.2.: Energy consumption of different implementations of PCG solver for G3_CIRCUIT [AHA+11].

We observe that, for the CG as well as for the preconditioned variant, the GPU implementation outperforms all CPU solvers by a large factor. This is true for the execution time as well as for the energy need. For the latter parameter, the improvement is smaller, since the power consumption of both the CPU and GPU have to be taken into account. For the CG solver, the optimal CPU-based configuration (use of 4 threads) results in more than 5× higher execution time and about 4× higher energy consumption. For the preconditioned variant, where the additional matrix-vector multiplication in every iteration loop is payed off with a lower number of iterations, the speedup and energy saving derived from the use of the GPU as coprocessor are even higher. The optimal CPU implementation can at most reach 1/7 of the GPU performance and consumes more than 480% more energy.

Although most scientific codes target clusters with general-purpose processors, this test validates the assumption that the use of GPUs for elementary kernel operations may improve the overall performance of parallel scientific applications. Despite the additional initialization process and data transfer to the GPU, the high number of computing cores on a graphics processor compensates these overheads, and enables the GPU to perform parallel instructions faster and with higher energy efficiency [AHA+11].

5.4.2. Experiments on Dynamic Voltage and Frequency Scaling (DVFS)

In the next experiment we analyze the total energy consumption of a CG solver applied to a linear equation problem associated with the test matrix A318 (see Appendix B). We are especially interested in the effect of applying Dynamic Voltage and Frequency Control. Therefore, we apply implementations using different clock frequencies and furthermore differ between CPU implementations using 1,2,4 and 8 cores, and a GPU implementation outsourcing the matrix vector operations to a Tesla C1060 board (see C.4). The experiments were again conducted on the hybrid watt2-system located at the University of Jaume I (see C.2), and the results have also been published in [AHA+11].

hardware	CPU freq.	time	power/energy consumption		
			chipset	GPU	total
	$[MHz]$	[sec]	P_{avg} $[W]$	P_{avg} $[W]$	$[Wh]$
CPU 1T	2,000	2059.69	116.78	-	66.81
CPU 1T	800	3400.64	103.50	-	97.75
CPU 2T	2,000	1708.31	120.30	-	57.08
CPU 2T	800	2196.63	105.60	-	64.44
CPU 4T	2,000	1441.78	123.99	-	49.66
CPU 4T	800	1674.62	108.11	-	50.29
CPU 8T	2,000	1395.37	129.33	-	50.13
CPU 8T	800	1481.48	110.46	-	45.45
GPU	2,000	253.22	149.04	61.89	14.84
GPU	800	254.25	138.50	61.45	14.12

Table 5.3.: Energy Consumption of different implementations of CG solver for A318 [AHA+11]. Results are labelled as "CPU nT" where "n" equals the number of threads / cores.

In Table 5.3 we report the execution time, average power demanded by the chipset and GPU (P_{avg} in average Watts $[W]$) and total energy consumption of the codes when DVFS is employed to set the operating frequency of the AMD cores to 2.00 GHz and 800 MHz for the time of the CG solver. Since rescaling the CPU frequency is on one side only beneficial when applied to all cores, and introduces some overhead on the other side, additional rescaling inside the algorithm may not improve the overall performance [AHA+11].

The results show that, for the CPU-based codes, there is no direct correlation between computation time and the total energy cost. Reducing the CPU-frequency using DVFS lowers the power consumption of the CPU, but since less operations can be conducted per unit of time, it increases the computation period. In the end, lowering the frequency does not necessarily yield an improvement to the energy cost. This may stem from the fact, that the cores are only one energy consumer among a long list of devices, that may not necessarily benefit from lowering the clock rate [AHA+11]. Using more cores reduces the execution time, but increases the energy consumption. Since not only the cores are consuming but also the memory and the chipset demand for some power, whether increasing the number of cores pays off depends on the application and the memory bandwidth. Only if the application is CPU-bound increasing the number of cores reduces the energy consumption; for memory-bound operations this may not be true in general [AHA+11]. On NUMA architectures, like the one employed in the experiments, using a large number of cores increases also the memory bandwidth, which may lead to a different ratio between the memory- and CPU workload. This effect can be observed when 8 threads are used.

Furthermore, we observe the glaring superiority of the GPU-accelerated implementations. The high computational power provided by the graphics processing units can be leveraged for the expensive matrix-vector operations, causing speedups of up to five. When utilizing the GPU as coprocessor, using DVFS is beneficial, since the operations conducted by the CPU are few and do not demand a high operation frequency [AHA+11].

Finally we observe that the graphics card consumes some power over the PCIe. Although it is in the used measurement setup not possible to provide explicit numbers for this energy transfer, the specifications do not allow a higher throughput than 75 Watts via PCIe[4].

Using the GPU-implementation for this test case, the improvement gained by using DVFS counts up to at most 5.6% of the total energy consumption (see last column of the GPU implementations in Table 5.3). The reason for this moderate result is that calling a GPU kernel operation sets the CPU into a busy-wait, a mode where the host system is waiting for feedback from the kernel, sending steadily requests to the device [AHA+11].

5.4.3. Experiments on Idle-Wait

In [AHA+11] also the effects of the idle-wait technique we introduced in Section 5.3.3 were analyzed. In this Section we will report the experimental results provided in [AHA+11]. While we still stick to the matrix system A318, we report the average power consumption of the GPU-accelerated CG solver in different implementations.

Figure 5.4 illustrates the power demand of different energy-saving techniques applied to the CG solving process of the linear system A318 during a period of 12.5 seconds, chipset-measurement only. Here, using DVFS to lower the operating frequency of the CPU cores from 2.0 GHz to 800 MHz (line labeled as "CG+DVFS") does not affect the iteration time, as most of the computations are performed on the GPU. The results show that with DVFS alone the improvement is small; the `nanosleep()` function yields a certain drop of the power consumption (line "CG+idle-wait"); and the combination of both techniques improves the performance further (line "CG+DVFS+idle-wait").

Based on these results on the average power consumption, we now want to analyze the improvement with respect to the total energy consumption gained by the idle-wait technique. Therefore, we employ three GPU-implementations of the CG solver and its Jacobi-preconditioned derivative [AHA+11]:

[4]PCI-SIG; `http://www.pcisig.com/specifications/pciexpress/`

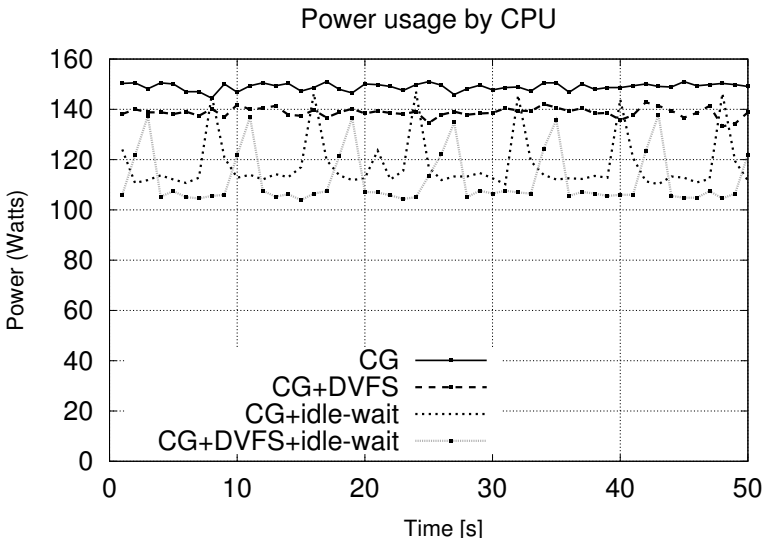


Figure 5.4.: Power consumption of different energy-saving techniques applied to the CG-solver, chipset measurement. [AHA+11]

(i) The first implementation is straight-forward, without DVFS or any other power-saving technique. The CPU of the host system runs at full speed (2.0 GHz) during the complete solving process.

(ii) Using DVFS, we scale down the frequency of the host system to 800 MHz during the operation of the GPU-accelerated solver. This has the drawback of slow CPU computations for this part, but since these computations are minor, this choice seems reasonable.

(iii) Additionally to DVFS we set the host system to sleep for the time the GPU performs the sparse matrix-vector multiplication.

Tables 5.4 and 5.5 collect the results obtained with the (i)–(iii) solver implementations, applied to selected linear systems described in Appendix B, using, respectively, a plain CG solver and a Jacobi-preconditioned one. We measure the total energy consumption by adding the energy use of chipset and GPU. The last two columns in both tables reflect the improvement in power consumption that can be obtained by using DVFS and the combination of DVFS and idle-wait.

matrix	energy consumption $[Wh]$			improvement [%]	
	(i)	(ii)	(iii)	(i)→(ii)	(i)→(iii)
A318	14.84	14.12	12.18	5.1	21.8
APACHE2	1.98	1.99	1.82	-0.5	8.8
AUDIKW_1	no convergence			-	-
BONES10	no convergence			-	-
ECOLOGY2	2.30	2.27	2.09	-1.3	10.0
G3_CIRCUIT	11.48	11.11	10.10	3.3	13.7
LDOOR	no convergence			-	-
N24K	26.43	25.42	21.17	3.97	24.8

Table 5.4.: Energy consumption of different implementations of the CG solver, chipset + GPU [AHA+11].

While for some problems only the preconditioned variant of the CG solver converges, applying a preconditioner is not always reasonable. There exist problems where the plain

matrix	energy consumption [Wh]			improvement [%]	
	(i)	(ii)	(iii)	(i)→(ii)	(i)→(iii)
A318	14.84	14.12	12.18	5.1	21.8
APACHE2	1.75	1.76	1.64	-0.6	6.7
AUDIKW_1	47.98	45.61	38.15	5.2	25.8
BONES10	157.32	150.16	125.78	4.8	25.1
ECOLOGY2	2.51	2.45	2.29	2.4	9.6
G3_CIRCUIT	2.71	2.63	2.38	3.0	13.9
LDOOR	43.22	41.18	34.79	5.0	24.2
N24K	34.62	32.97	27.64	5.0	25.3

Table 5.5.: Energy consumption of different implementations of the PCG solver, chipset + GPU [AHA+11].

implementation is superior, but for most systems, adding a preconditioner improves the performance. At this point, it is worth mentioning that we only evaluated example A318 without the preconditioner. The reason for this is twofold. First, applying a Jacobi-preconditioner to this system does not improve the convergence behavior. Second, the additional memory required for the preconditioner poses a problem for our system equipped with only 4 GB of GPU-memory on the Tesla C1060 board (see Appendix C.4). The obtained results demonstrate that DVFS alone renders only small improvement to the power consumption, and in some cases it even triggers a higher energy cost. This happens when the time and the related energy overhead triggered by rescaling the CPU frequency exceeds the power savings.

Applying the combination of DVFS and the idle-wait, we observe an improvement in the power consumption for all test-cases. Still we appreciate large differences in the scale of saving. While for some systems the energy saving is in the range of 1/4, it only sums up to a few percent for some others. There exist two main factors determining the energy savings [AHA+11]:

1. The time of the matrix-vector operations conducted by the GPU dictates whether it is reasonable to sleep the host system for a considerable time-frame. If the overhead due to calling idle-wait exceeds the execution time of the GPU kernel, no improvement can be obtained. Additionally, the sleep function takes some time to scale down the energy consumption of the processor. For the used system, the average time for the power to settle for the respective values when activating (deactivating) idle-wait approximates 50 (74) microseconds.

2. The sparsity pattern of a matrix determines the ratio between the cost of a sparse matrix-vector product (BLAS-2 [FoE10]) and a vector operation (BLAS-1 [FoE10]). Since for the dimensions of the systems that were evaluated the execution of the BLAS-1 kernels on the GPU usually took less time than the overhead for calling idle-wait, we set the host system to sleep only for the sparse matrix-vector product. Hence, the improvement comes from the GPU kernels conducting the matrix-vector operations. If these account for a large part of the overall computation time, we can expect notable energy savings.

The first point leads to the conclusion that the usage of DVFS and idle-wait in the context of GPU-accelerated algorithms is only reasonable for systems with expensive matrix-vector products. For other problems, either the computational cost of solving the linear system is low or the condition number is very high, leading to a large number of iterations. In both cases, using the CPU with less computing cores but working at a higher frequency than the GPU leads to the acceleration of the solver. Hence, for the general case where

the GPU-implementation of a solver is superior, the dimension of the system allows an efficient use of DVFS and idle-wait.

The second point suggests that larger benefits could be expected from the application of these techniques to the solution of dense linear systems via iterative methods.

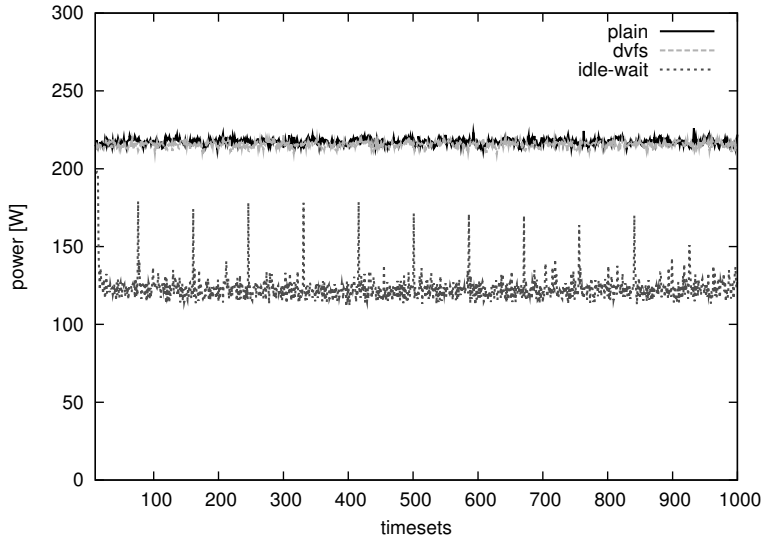
5.4.4. Asynchronous Iteration for Energy-Efficient Numerics

As we have seen in the last sections, applying DVFS to reduce the power demand and the overall energy consumption especially pays off for algorithms including long computing routines, where the processor frequency can be optimized to the demands. Adapting the processor voltage and frequency regularly during an application execution triggers some runtime overhead which may again cause a rising energy consumption. Very similar, using idle-wait only is beneficial for cases where the overhead associated with calling idle-wait is compensated for by the lower power usage during the GPU kernel routine. When using highly parallel or even heterogeneous hardware, the length of the distinct routines is usually limited by the synchronization points between the different processors. In case of classical iterative solvers, these usually occur at latest in between the iterations (e. g. Jacobi, see Section 2.4.1) or even earlier to compute a parameter (e.g. Conjugate Gradient, see Algorithm 7). Asynchronous iteration algorithms we investigated in Chapter 3 and 4 minimize the synchronization points: Since even the residual computation for checking the stopping criterion can be handled independently and without synchronizing the iteration process, the algorithm may run asynchronously until the preset stopping criterion is fulfilled. For this reason, asynchronous iteration algorithms are suitable targets for energy saving mechanisms like DVFS and idle-wait. While the power advantage using asynchronous iteration is apparent as idle-wait can be applied for the complete algorithm execution, the energy efficiency is determined by the trade-off between convergence, overall runtime and power demand. For setups where switching to asynchronous iteration decreases the overall runtime, also the needed energy consumption is reduced. But also for cases where the slower convergence of asynchronous iteration increases the runtime, the lower average power dissipation due to the absence of synchronization may trigger a lower total energy expenditure of the iteration process. In order to analyze this issue, we conduct experiments on the energy and power profile of asynchronous iteration, and compare the respective results with those obtained from the synchronized Jacobi counterparts.

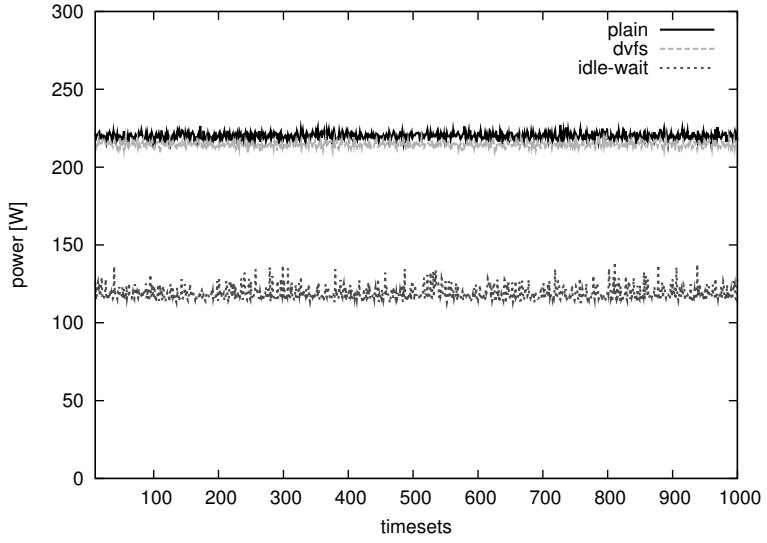
Since component wise relaxation methods rarely feature any parameter computations, they can leverage idle-wait in general more aggressively than their Krylov-subspace counterparts. For the asynchronous iteration, even the synchronization between the iteration steps is ignored, which allows merging multiple iterations into one kernel. Thus, while for the Jacobi method the ratio between the length of the kernels updating the components and the length of the synchronization using the CPU is quite high, the asynchronous algorithm allows the application of idle-wait for a large set of iterations. In fact, the number of iterations merged into one kernel is only limited by the instruction buffer of the GPU. For this set of iterations, idle-wait can be applied to decrease the CPU power demand.

The first experiment, again conducted on the watts2-system (see Appendix C.2), is dedicated to monitor the power consumption of Jacobi and asynchronous iteration when applying different energy saving mechanisms. In Figures 5.5a and 5.5b we report the CPU power demand of the synchronous Jacobi and asynchronous iterations, respectively, in plain mode, using DVFS or idle-wait. The linear system we target in this analysis is the matrix FV3, see Appendix B.

One first result to note in the figures is the small improvement attained from the application of DVFS to both the Jacobi and asynchronous iterations (line labelled as `dvfs`) when compared with the respective solver running at the highest frequency (line labelled as



(a) Jacobi



(b) async-(5)

Figure 5.5.: Power dissipated by the CPU for the Jacobi iteration and the async-(5) method, respectively featuring different energy saving techniques. The timesets denote the respective samples using a frequency of 25 Hz.

`plain`). While scaling down the processors' frequency from 2.0 GHz to 800 MHz reduces the CPU power demand by approximate 7% for the asynchronous iteration (see Figure 5.5b), and a mere 5% for the Jacobi method (see Figure 5.5a), when taking the GPU power draft into account, the improvements become mostly negligible.

On the other side, applying idle-wait indeed improves the power profile considerably (line labelled as `idle-wait`). For the Jacobi method, we can observe in Figure 5.5a a decrease in the CPU demand by 43%, taking the GPU into account (we refrain from adding this data to Figure 5.5 for clearness), the improvement still approximates 29%. The achievements stem from the excellent ratio between the GPU kernel runtime and the synchronization overhead including the CPU.

For the asynchronous iteration, the benefits of applying idle-wait are slightly higher (see Figure 5.5b). The reason is that, in case of very high iteration numbers, the only task handled by the CPU is to stack the next set of operations into the instruction buffer of the GPU (due to the limited size of the device buffer). Exploiting idle-wait in the

asynchronous iteration decreases the CPU power consumption by 44% and, taking the complete CPU-GPU system into account, the power demand can be cut by 33%. In contrast with Krylov subspace methods, where the ratio between the kernel times and the CPU-handled operations determines the average power savings [AHA+11, ACF+], in component-wise relaxation methods the savings are almost independent of the linear equation system. This is due to the fact, that component wise relaxation algorithms lack scalar-products or parameter computations interrupting the kernel routines.

We now focus on the energy saving attained by these techniques. Table 5.6 reports the computation time to converge (in seconds, [sec]), the average power demand (P_{avg}, in Watts [W]), and the energy consumption (E_{tot} in Watts-hours [Wh]) of the Jacobi and asynchronous iterations. In this experiment, we also include a third solver variant, that combines mixed precision iterative refinement (MPIR, see Section 2.3) with asynchronous algorithm as error correction solver in single precision. (We analyzed the combination of block-asynchronous iteration and mixed precision iterative refinement with respect to runtime in Section 4.12.1.)

The results reveal, that the block-asynchronous iteration outperforms the runtime of the Jacobi method in terms both of runtime and overall energy consumption of the iteration process. While the reduced computation time translates linearly into energy savings, the slightly lower average power consumption due to the more effective utilization of idle-wait leads to further benefits. Embedding the asynchronous iteration into the mixed precision iterative refinement framework triggers, at least for this test case, an additional reduction of the runtime as well as the energy consumption, in spite of the higher average power demand due to the extra computations that come with the parts of the MPIR framework making synchronizations necessary (see Section 2.3).

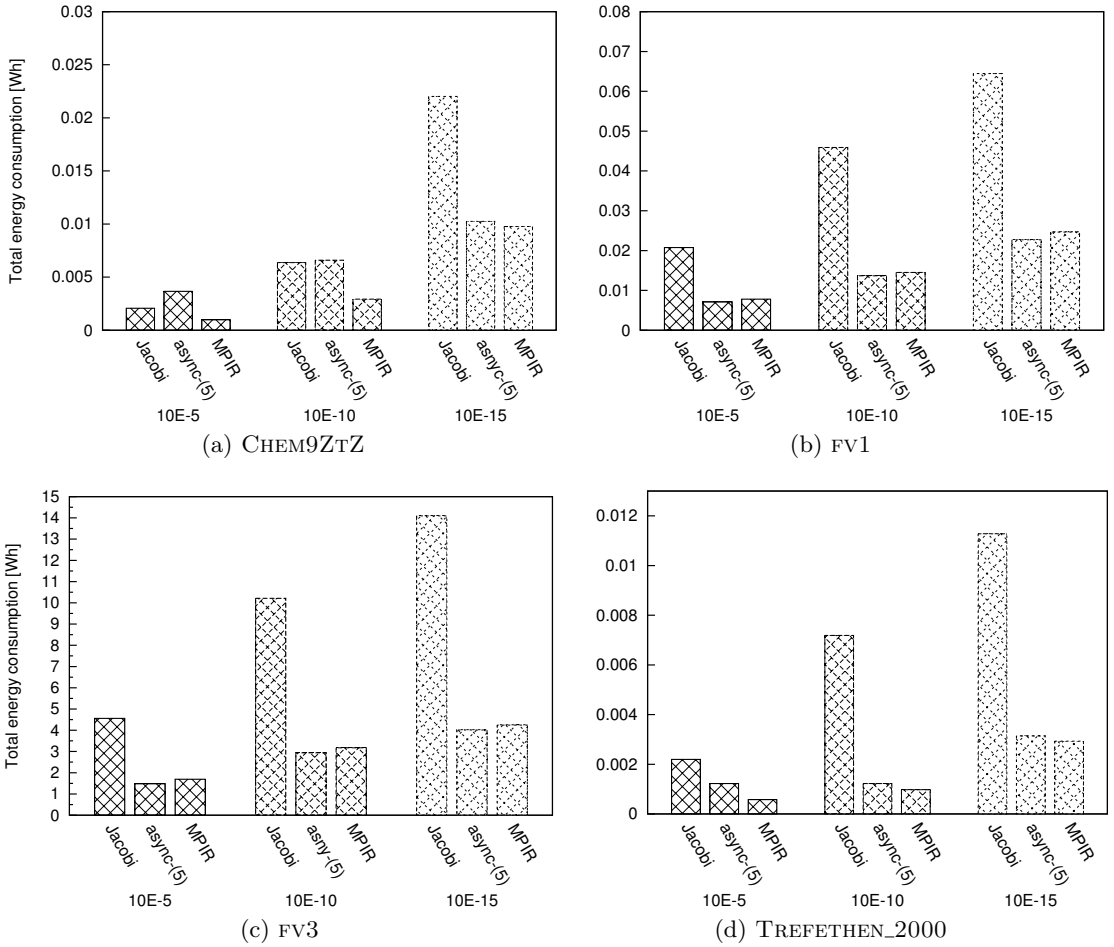
method	Time [sec]	P_{avg} [W]		E_{tot} [Wh]	
		plain	idle-wait	plain	idle-wait
Jacobi	406.10	217.60	124.85	24.35	14.08
async-(5)	133.09	216.72	118.98	7.98	4.38
Gain w.r.t. Jacobi	67%			67%	69 %
MPIR async-(5)	67.69	232.41	129.24	4.37	2.43
Gain w.r.t. Jacobi	83%			82%	83%

Table 5.6.: Runtime and energy characteristics of the different solver implementations. The power and the energy results are the aggregated CPU and GPU demands covering the time to convergence.

Finally, we aim for comparing the energy improvements for a larger set of test systems. For this purpose we report the total energy consumption of the Jacobi method, and the block-asynchronous iteration in plain and MPIR mode featuring idle-wait.

For the test system Chem9ZtZ, we observe in Figure 5.6a the superiority of the Jacobi method over the async-(5) method for low accuracy approximations. For higher accuracy, the reduced overall runtime triggers also an energy superiority of the async-(5) implementation. Embedding the method into the MPIR framework is for this experiments setup superior for all accuracy demands. This is different when targeting the system fv1 (see Figure 5.6b): Similar to analyzing the runtime performance in Figure 4.33 (see results for the system C1060 in this plot), the overhead associated with the initialization process of MPIR can not be compensated for.

For the test cases fv1 and fv3 the results in Figure 5.6b and 5.6c are very similar to the runtime analysis in Figure 4.33. The async-(5) energy dissipation is only a fraction of the Jacobi energy demand, but embedding async-(5) into the MPIR framework is not beneficial. While the energy dissipation is smaller than for the Jacobi implementation due to the considerably decreased runtime, the additional synchronizations cause a higher



(a) Chem9ZtZ (b) fv1

(c) fv3 (d) Trefethen_2000

Figure 5.6.: Total CPU energy consumption for the different test cases using different relative residual stopping criteria. The implementations are Jacobi, async-(5) and mixed precision iterative refinement based on an async-(5) error correction solver.

average power draft that can not be compensated for. For the Trefethen_2000 test case (see Figure 5.6d), the MPIR variant is superior to the plain async-(5) implementation, but the improvements become smaller for high accuracy approximations. In the comparison with the synchronized Jacobi, the block asynchronous methods are especially beneficial when aiming for high accuracy solution approximations.

When comparing the energy consumption results in this section with the runtime improvement when switching from async-(5) to the MPIR framework (see the C1060 results in Figure 4.33), we note that for the small test cases Chem97ZtZ and Trefethen_2000 the energy reduction is smaller than the acceleration. This stems from the fact that the MPIR framework includes synchronization points for the residual computation and the iteration update that limit the length of the idle-wait periods. For the larger systems, especially for high condition numbers implying high iteration numbers, these effects are small.

Summarizing the experimental results, we may conclude that while already the improved runtime performance makes asynchronous methods attractive when aiming for a reduced energy consumption, the algorithms' design allows a very efficient usage of power-saving techniques provided by the hardware.

6. Summary

The main contribution of this thesis is the derivation, theoretical and experimental analysis of block-asynchronous iterative methods suitable for hardware systems accelerated by graphics processing units.

The work is motivated by the recent hardware shifts and the challenges that have to be addressed to the further progress in scientific High Performance Computing.

We first gave a short survey on existing iterative methods with an emphasis on their suitability for High Performance Computing. This included the topics of mixed precision iterative refinement methods and blocking strategies enabling higher parallelization. We then reviewed the theory of asynchronous relaxation methods, providing a classification of the different schemes and existing convergence results that we later extended for our purposes.

After introducing the block-asynchronous iteration method that includes local iterations on subdomains chosen adequate for efficient GPU implementations, we analyzed its convergence and fault-tolerance properties. We experimentally investigated the non-deterministic behavior for GPU-based implementations of block-asynchronous iteration and compared the performance to synchronized methods. The results revealed that the higher parallelization potential allowing for a more efficient hardware usage may overcompensate the inferior convergence properties of asynchronous algorithms, so that they can outperform the traditionally applied schemes. Investigating multi-GPU implementations using peer-to-peer communication between the distinct devices, we observed that the communication bottleneck imposed by the PCI-connection and often limiting the application performance can efficiently be overcome by asynchronous data transfers, which is possible due to the algorithm design.

We then aimed for adapting the block-asynchronous iteration to a specific problem. While the optimization for sparse linear systems of equations is straight-forward, accounting for the characteristics of the underlying problem offers a wide range of possibilities. For this purpose we introduced weighting techniques that take into account the matrix parts that are not considered in the local iterations. The convergence theory we derived was also reflected in the experimental results on the different techniques: The use of weights can improve convergence as well as performance significantly. Furthermore, we demonstrated that the possibility of choosing the subdomain size consistent to the discretization allows the adaptation of block-asynchronous iteration to the discretization of a partial differential equation.

Since relaxation methods often provide important contributions as smoother in multigrid methods, we investigated the potential of replacing the traditionally employed synchronous smoother by block-asynchronous iteration. Experimental results revealed that, especially for highly parallel and heterogeneous systems, significant performance improvement can be obtained.

In the context of mixed precision iterative refinement methods, we first analyzed the suitability of block-asynchronous iteration as a low precision error correction solver. Experiments for different problems and using different GPU architectures revealed that performance improvements strongly depend on the properties of the problem and the hardware characteristics like cache size and FLOP rate in the different precision formats. We then designed a block-asynchronous iterative method where also the residual computation is handled asynchronously. After a theoretical convergence analysis we investigated the trade-off between synchronous and blockwise asynchronous residual computation experimentally. A strong dependency of the performance on the problem characteristics was also observed for this implementation.

Finally, we demonstrated that block-asynchronous iteration can also efficiently be applied when solving nonlinear differential equations. This was achieved by illustrating experimental results obtained with the iterative method in a pattern formation simulation arising in mathematical biology.

We then addressed the topic of energy-efficiency in scientific High Performance Computing and experimentally revealed the high energy efficiency of block-asynchronous iteration which stems from a very efficient utilization of the power saving mechanisms provided by the hardware.

Since we already had identified the scalability and a high tolerance to communication latencies and hardware failure, the proposed algorithm efficiently combines the most important properties necessary to address the challenges associated with exascale computing. Although the problem of exascale numerics was in this thesis reduced to node-level, the properties of asynchronous methods imply the suitability also on parallel systems. For this reason, the use of block-asynchronous iteration may be considered as an adequate algorithm for emerging technologies with much higher levels of hardware concurrency.

Future research on this topic should not only address the efficient implementation of block-asynchronous iteration on emerging technologies like Intel's MIC, new generations of GPUs, FPGAs etc, but also focus on the theoretical properties of the method. Particularly, fundamental knowledge about the dependency between the partial differential equation, the discretization, the solver parameters and the problem data may yield remarkable improvements in performance.

Appendix

A. Implementation of Numerical Algorithms

A.1. Floating Point Formats

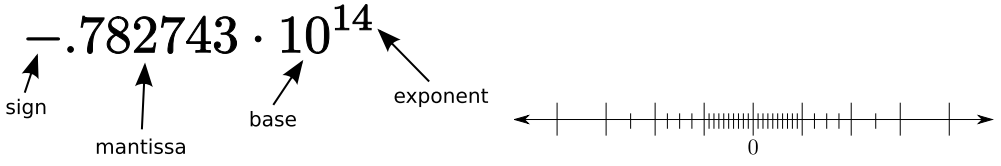
To enable any machine to perform computations, the used numbers have to be represented intrasystem [Kul08]. In numerical computing, *floating point* describes a system for a numerical representation in which a string of digits represents a rational number. The term floating point refers to the fact that in this representation, the radix point is placed relative to the significant digit, meaning the first counting digit.

Despite the existence of different floating point formats, they all use the same principle to represent a real number r. All floating point numbers consist of a sign bit s, a mantissa m, a base b and an exponent e.

- **sign bit:**
  The sign bit $s \in \{-1, 1\}$ gives the sign of the number r.
- **mantissa:**
  The mantissa m represents the digits of the number r. The more digits are stored in the mantissa, the higher the accuracy of the representation of the number r. To get an overall standard, one usually norms the mantissa such that it has the decimal point after the first digit
- **basis:**
  For scientific use, one normally chooses the basis $b = 10$, but as computers use the binary format, the computer floating point formats have the basis 2.
- **exponent:**
  The exponent e contains the information of the floating point.

A rational number r is represented in the form $r \approx s \cdot m \cdot b^e$, where we want to stress that not all rational numbers can be represented correctly, but as an approximation [Kul08].

An important fact about these floating point systems is that the representable numbers do not all have the same distance. This means, that the higher the exponent of a number is,



(a) Floating point number in the decimal system.

(b) Representable numbers in a floating point format [Kul08].

Figure A.1.: Visualization of a floating point format and the within representable numbers.

the higher the distance is between two numbers that can be represented, see Figure A.1b. Therefore, one uses the *minimal relative distance* to characterize every format, while half the minimal relative distance is usually denoted with the *machine accuracy*. Each floating point format is then characterized by the machine accuracy and the smallest number $\varepsilon \neq 0$ that can be represented.

A.2. IEEE754

One of the standards for binary floating point arithmetic is the IEEE754 [Kul08]. It became the most widely-used standard for floating-point computation, and is implemented in nearly all available systems. It includes correct rounding and rules for subnormal numbers and $\pm\infty$ (Section A.3). In the IEEE 754 standard, the two main floating point numbers are denoted as *single precision* and the *double precision*. Both consist of a sign bit, an exponent and a mantissa represented in the binary system, and both formats use the basis 2. In case of single precision, the whole floating point number consists of 32 bits, in the case of double precision 64 bits. We are not interested in the detailed configuration of these bits, however it is useful to have knowledge of the smallest absolute value ε and the machine accuracy δ, that characterize the formats.

type	size	exponent	mantissa	ε	δ	digits
single	32 bit	8 bit	$23+1$ bit	$1.2 \cdot 10^{-38}$	$5.69 \cdot 10^{-8}$	$6-9$
double	64 bit	11 bit	$52+1$ bit	$2.2 \cdot 10^{-308}$	$1.11 \cdot 10^{-16}$	$15-17$

Figure A.2.: Comparing Single and Double Precision Standard IEEE 754 [Kul08, Kah96, MBdD$^+$09].

For normalized floating point numbers in IEEE 754, the first bit is predefined, and therefore needs not to be stored. For this reason, accounting for this *hidden bit*, the total length of the mantissa is for normalized numbers increased by one [Kul08].

Aside from these standard single and double formats, there exist different extended versions of these number formats, however we focus on these two possibilities, since most systems support standard single precision and standard double precision IEEE 754.

If not otherwise denoted, these formats are used for all implementation within this thesis.

Performing computations in single precision or double precision will result in different computational cost. In most cases, the size difference between the double precision format and the single precision format leads to a speedup factor of 2 when switching from double to single precision.

For some systems, especially when only single precision arithmetic is supported by the hardware, larger speedups can be observed. Then, double precision computations are only possible by using tricks, such as splitting one double precision number into two single precision numbers. This often results in poor double precision performance.
In "Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid" [GS11] Dominik Göddeke and Robert Strzodka evaluate the speedup for different architectures, and observe that for NVIDIA's GT200 chip the factor is around eight (see Table C.4). For other architectures including the AMD's R800 (Evergreen) chip and the first generation Cell processor they observe speedups of five, respectively 7 when executing single precision floating point instead of double precision operations [GS11]. However, in many applications, the theoretical peak performance is not the relevant factor however,

because the computations are memory-bound (memory wall problem). The use of single precision consequently halves the bandwidth requirements of a given computation, and one can expect up to a twofold speedup [GS11]. Nevertheless, the savings from using single precision are often even higher in practice, because the argument applies to all levels of storage: Single precision puts less pressure on the registers and twice the amount of data can be held in small, fast on-chip memories and caches, resulting in a better data reuse and, in particular in CUDA, a higher occupancy of multiprocessors [GS11].

A.3. Floating Point Arithmetic

We want to provide some background on the impact of the floating point format's characteristics when performing computations.
The fact that only a small amount of real numbers can be displayed correctly in a certain floating point format, implies that most representations in a certain format contain rounding errors. In this case, the *relative representation error* [Dem97] is

$$\frac{|r - r_{float}|}{|r|} \leq \delta.$$

When performing a computation $a \odot b$, where $\odot$ is one of the binary operations ($+$, $-$, $\cdot$, $/$), the solution can usually not be represented exactly in the used floating point format. In this case, it must be approximated by a nearby floating point number before it can be stored in memory or register [Dem97]. If we denote this approximation with $fl(a \odot b)$, the difference $|(a \odot b) - fl(a \odot b)|$ is called *round-off error*. We say the arithmetic *rounds correctly* [Dem97], if $fl(a \odot b)$ is the nearest floating point number. In the case of two floating point numbers having the same distance to $fl(a \odot b)$, there exist different possibilities to define a convention. In the IEEE 754 format it is defined that in this case, the last digit is rounded to even [Kah96, KMP11]. When rounding correctly, we can write

$$\frac{|fl(a \odot b)|}{|a \odot b|} \leq (1 + \delta),$$

where δ denotes the machine accuracy.
Furthermore, most floating point formats contain symbols and rules for *subnormal numbers*, i.e. unnormalized floating point numbers with the minimum possible exponent, division by zero, and computations concerning $\pm\infty$.

Performing a computation in different floating point formats will residue in different computational cost. This is due to the fact that the number of arithmetic operations necessary for a computation depends on the length of the mantissa. Hence, performing a computation in a lower precision format is usually faster than performing it in a higher precision format.

B. Linear Systems of Equations

The experiments in this thesis utilize different matrices. For most experiments, the matrices are taken from the University of Florida Matrix Collection[1].

The matrix properties and sparsity plots can be found in Table B.1 and Figure B.3.

The first matrix, A318 is derived from a finite difference discretization of the 3D Laplace problem.

[1] University of Florida Matrix Collection (UFMC); `http://www.cise.ufl.edu/research/sparse/matrices/`

APACHE2, AUDIKW_1, LDOOR, ND24K and S1RMT3M1 derive from finite element discretizations of structural problems. BONES10, part of the Oberwolfach matrix group, is the coefficient matrix related to the finite element discretization of a bone [2]. The matrix, CHEM97ZTZ, comes from statistics [3]. ECOLOGY2 is a landscape ecology problem, using electrical network theory to model animal movement and gene flow. Using a 2D, 1000-by-1000 mesh (5 pt stencil) the obtained coefficient matrix is symmetric and positive definite. (Source: Brad McRae, National Center for Ecological Analysis and Synthesis Santa Barbara, CA.) Matrices FV1 and FV3 are finite element discretizations of the Laplace equation on a 2D mesh. Therefore, they share a common sparsity structure, but differ in dimension and condition number due to the different finite element choice. While G3_CIRCUIT has its origins in a circuit simulation, ND24K derives from a 2D/3D problem. The matrix TREFETHEN_2000 [Tre02] is a 2000×2000 matrix where all entries are zero except for the ones at the positions (i, j) where $|i - j| = 2, 4, 8, 16 \ldots$. Furthermore, the main diagonal is filled with the primes $2, 3, 5, 7, 11 \ldots 17389$. Hence, this matrix has many off-diagonal entries distributed over the diagonals that are by a power of 2 distant to the main diagonal. The matrix TREFETHEN_20000 is generated in the same pattern, only the size is increased by the factor of 10.

Matrix name	Description	$\#n$	$\#nnz$
A318	3D Laplace	32,157,432	224,495,280
APACHE2	structural problem	715,176	4,817,870
AUDIKW_1	structural problem	943,695	77,651,847
BONES10	3D trabecular bone	914,898	40,878,708
CHEM97ZTZ	statistical problem	2,541	7,361
ECOLOGY2	2D problem	999,999	4,995,991
FV1	2D/3D problem	9,604	85,264
FV2	2D/3D problem	9,801	87,025
FV3	2D/3D problem	9,801	87,025
G3_CIRCUIT	circuit simulation problem	1,585,478	7,660,826
LDOOR	structural problem	952,203	42,493,817
ND24K	2D/3D problem	72,000	28,715,634
S1RMT3M1	structural problem	5,489	262,411
TREFETHEN_2000	combinatorial problem	2,000	41,906
TREFETHEN_20000	combinatorial problem	20,000	554,466

Table B.1.: Dimension and characteristics of the SPD test matrices.

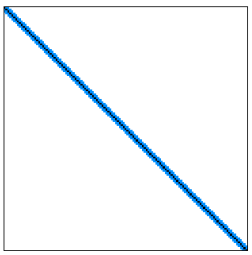
For some matrices, we additionally provide in Table B.2 some of the convergence related characteristics as well as of their corresponding iteration matrices. The motivation is, that especially these are used for the asynchronous iteration experiments, since they fulfill the sufficient convergence condition.

Unless otherwise stated, we take the number of right-hand sides to be one for all linear systems.

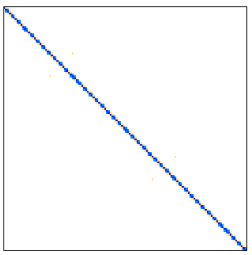
C. Hardware Platforms

Since the experiments in this paper were conducted on different hardware platforms, we want to provide an overview about the systems' configurations.

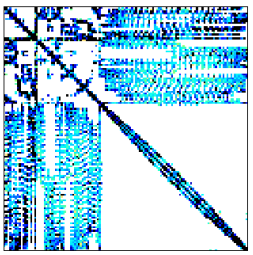
[2] For more details see `http://www.cise.ufl.edu/research/sparse/mat/Oberwolfach/README.txt`
[3] For more details see `http://www.cise.ufl.edu/research/sparse/mat/Bates/README.txt`



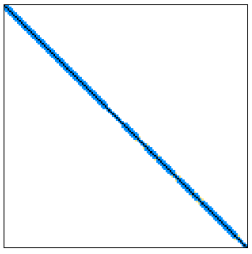
(a) A318



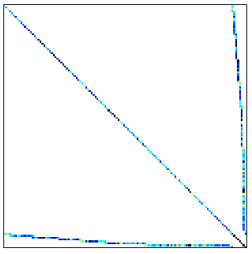
(b) APACHE2



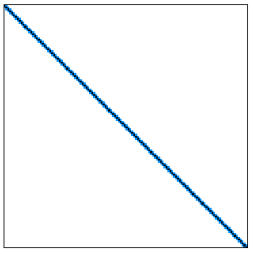
(c) AUDIKW_1



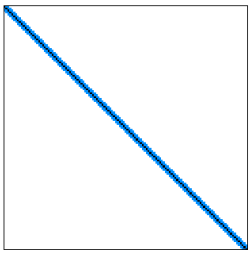
(d) BONES10



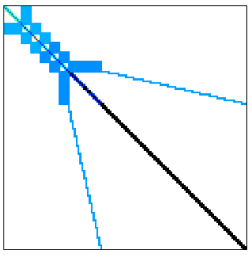
(e) CHEM97ZTZ



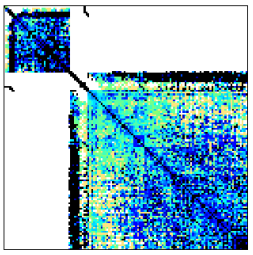
(f) ECOLOGY2



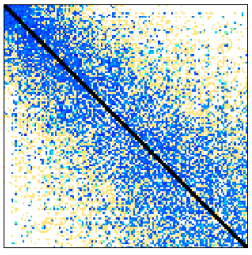
(g) FV, FV2, FV3



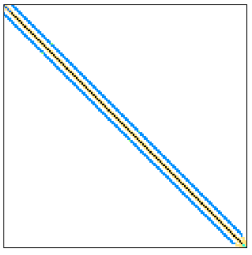
(h) G3_CIRCUIT



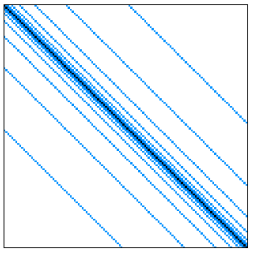
(i) LDOOR



(j) ND24K



(k) S1RMT3M1



(l) TREFETHEN_2000

Figure B.3.: Sparsity plots of SPD test matrices.

Matrix name	cond(A)	cond($D^{-1}A$)	$\rho(I - D^{-1}A)$
Chem97ZtZ	1.3e+03	7.2e+03	0.7889
fv1	9.3e+04	12.76	0.8541
fv2	9.5e+04	12.76	0.8541
fv3	3.6e+07	4.4e+03	0.9993
s1rmt3m1	2.2e+06	7.2e+06	2.65
Trefethen_2000	5.1e+04	6.1579	0.8601
Trefethen_20000			

Table B.2.: Convergence characteristics of selected test matrices and of their corresponding iteration matrices.

from \to	host	GPU0	GPU1	GPU2	GPU3
host	16.0	5.94	5.93	5.91	5.91
GPU0	6.42	-	4.91	3.18	3.18
GPU1	6.41	4.91	-	3.18	3.19
GPU2	6.39	3.18	3.20	-	4.91
GPU3	6.39	3.19	3.20	4.91	-

Table C.3.: Inter-device memory bandwidth [GB/s].

C.1. Supermicro-System

The supermicro-system is a heterogeneous GPU-accelerated multicore system located at the Engineering Mathematics and Computing Lab (EMCL) part of the Karlsruhe Institute of Technology, Germany. Based on a X8DTG-QF mainboard, the system is equipped with two Intel XEON E5540 @ 2.53GHz and 192 GB main memory (dual-IOH architecture). In Figure C.4 we provide an overview about the mainboard configuration. The purpose is to give the reader a general idea about the architecture of modern hardware platforms. The system is accelerated by 4 Fermi C2070 (14 Multiprocessors x 32 CUDA cores @1.15GHz, 6 GB memory, see Appendix C.4). While the CPU interconnection is handled by QPI (up to 16 GB/s), always two GPUs are connected to one CPU through a PCI-ex16 (up to 8 GB/s). This leads to very specific performance characteristics. The very fast QPI allows for efficient inter-CPU communication, but when targeting multi-GPU implementations, GPU-direct can only be used for two GPUs connected to the same CPU as NVIDIA's GPU-direct does not support communication via the QPI [NVI09]. On the other hand, the full support of PCI-ex16 for all GPUs allows for very fast data transfers between the host and the different GPUs. This is especially interesting when using the asynchronous multicopy function allowing for the simultaneous communication with different graphic processors [NVI09]. Then, the limiting factor becomes the data locality in the main memory, since data located in the memory of the respectively other CPU leads to additional communication overhead.

In benchmarks testing the communication bandwidth we obtained a detailed data about the respective transfer rates (see Table C.3). Note that the host\host bandwidth describes the inter-CPU bandwidth of the Dual-IOH machine using QPI. The GPU \GPU bandwidth is either the GPU-direct bandwidth (if both GPUs are connected to the same CPU), or using the main memory as transfer storage. Analyzing the data, we observe, that the GPU-direct is not able to achieve the same memory bandwidth like the GPU-host communication. At this point we want to mention, that the results agree to investigations conducted by Saeed Iqbal and Shawn Gao [IG12, IGM12]

The GPU implementations are based on CUDA [NVI09], while the respective libraries

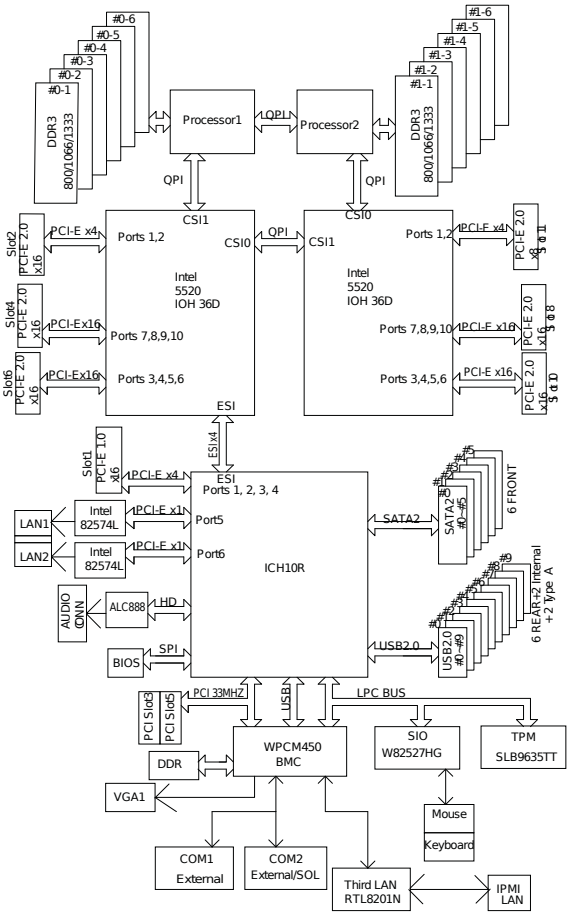


Figure C.4.: The Supermicro mainboard architecture [Sup10].

used are from CUDA 4.0.17 [NVI11].

The Supermicro System is embedded in a complex energy measurement setup based on the National Instruments Compact Rio[4]. Details allowing deep insight where and how power is measured are provided in the circuit plans in Figure C.5[5].

C.2. Watts-2

The Watts-2 platform consists of an AMD Opteron 6128 processor (8 cores) running at 2.0 GHz, with 12 MB of shared L3 cache and 24 GB of RAM. The double-precision peak performance of the multicore processor is 64 GFLOPS. The system is connected via PCIe (16x) to an Nvidia Tesla C1060 card with 4 GB of GDDR3 global memory. As main feature, the complete platform is embedded into a sophisticated power measurements setup (see Section 5.2.1).

C.3. Disco-System

The disco-system is a heterogeneous GPU-accelerated multicore system located at the University of Tennessee, Knoxville. The system's CPU is one socket Intel Core Quad Q9300 @ 2.50GHz and the GPU is a Fermi C2050 (14 Multiprocessors x 32 CUDA cores @1.15GHz, 3 GB memory). The GPU is connected to the CPU host through a PCI-e×16. The GPU implementations on this system are based on CUDA [NVI09], while the respective libraries used are from CUDA 4.0.17 [NVI11].

[4]National Instruments; `http://www.ni.com/`

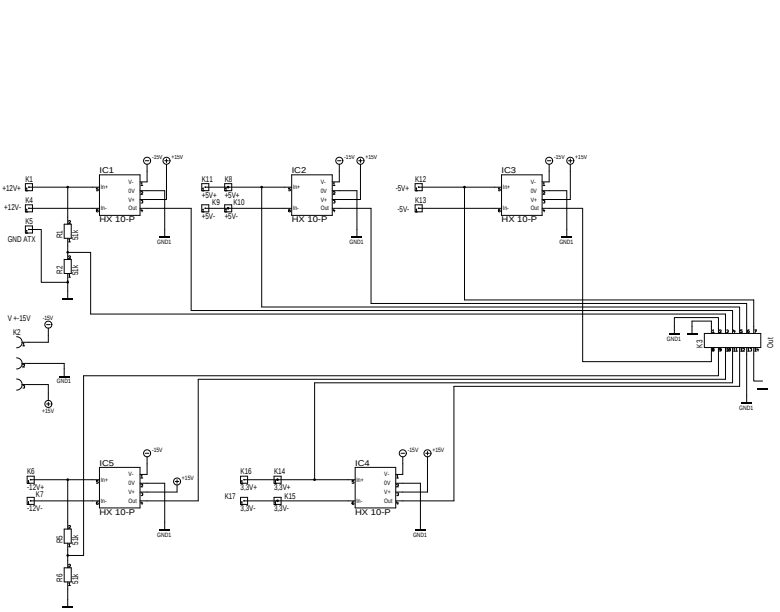
[5]I would like to thank Peter Schöck and Frederic Hupbauer for their efforts in the setup of the configuration.

Name	GTX280a	GTX580	Tesla C1060	Tesla C2070
Chip	GT200	GF110	T10	T20
Transistors	$1.4 \cdot 10^9$	$3 \cdot 10^9$	$1.4 \cdot 10^9$	$3 \cdot 10^9$
Core frequency	1.3 GHz	1.5 GHz	1.15 GHz	1.3 GHz
Thread Processors	240	512	240	448
GFLOPS (single)	933	1580	933	1030
GFLOPS (double)	78	790	78	515
Shared Memory/L1	16 KB	64 KB	16 KB	64 KB
L2 Cache	-	768 KB	-	768 KB
Memory	1 GB GDDR3	1.5 GB GDDR5	4 GB GDDR3	6 GB GDDR5
Memory Frequency	1.1GHz	2.0 GHz	0.8 GHz	1.5 GHz
Memory Bandwidth	141.0 GB/s	192.4 GB/s	102.0 GB/s	144.0 GB/s
ECC Memory	no	yes	no	yes
Power Consumption	236 W	244 W	200 W	190
IEEE double/single	yes/partial	yes/yes	yes/partial	yes/yes

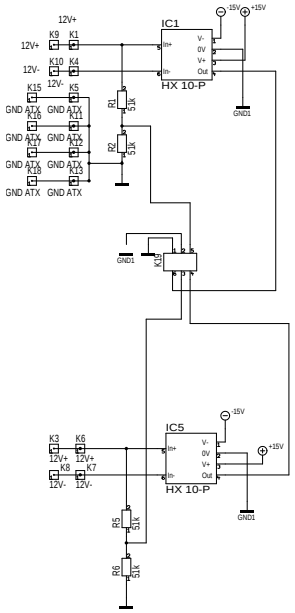
Table C.4.: Key system characteristics of the four GPUs used. Computation rate and memory bandwidth are theoretical peak values [NVI].

C.4. Recent GPU Architectures

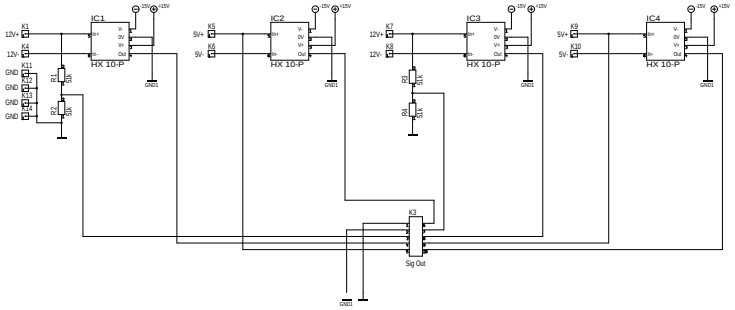
For the experiments in this thesis, GPU systems taken from the Fermi and the Tesla line of Nvidia were used. The C2070 and the C1060 are the server versions of the line, the GTX580 and the GTX280 are the consumer version, respectively. While the chip and on-board memory specifications are given in table C.4, the host system may have minor influence on the performance, since all computations are exclusively handled by the graphics. Note that the price for the larger (ECC protected) memory in the server versions is a lower memory bandwidth.



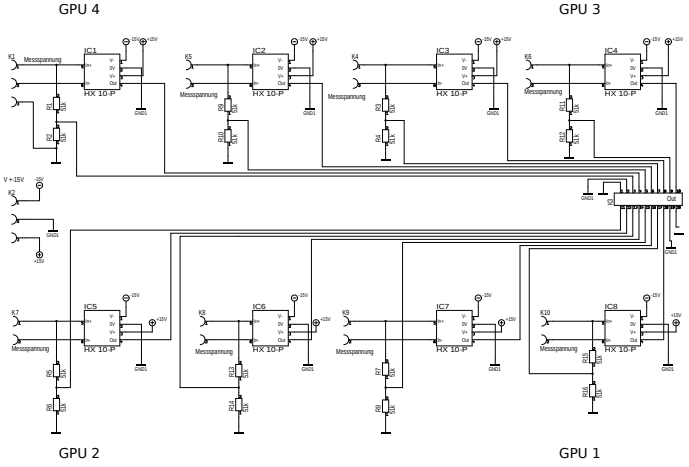
(a) ATX Power Supply.



(b) Mainboard.



(c) HDDs.



(d) GPUs.

Figure C.5.: Circuit diagrams for power measurement setup.

Bibliography

[ABC+] H. Anzt, A. Beglarian, S. Chilingaryan, A. Ferrone, V. Heuveline, and A. Kopmann, "A unified energy footprint for simulation software," *Computer Science - Research and Development*, pp. 1–8, 10.1007/s00450-012-0225-1.

[ABC+10] S. Ashby *et al.*, "The opportunities and challenges of exascale computing," Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, November 2010. [Online]. Available: http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf

[ABD+09] K. Asanovic *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.

[ACF+] H. Anzt, M. Castillo, J. Fernández, V. Heuveline, F. Igual, R. Mayo, and E. Quintana-Ortí, "Optimization of power consumption in the iterative solution of sparse linear systems on graphics processors," *Computer Science - Research and Development*, pp. 299–307.

[ADI+12] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, "Saving energy in the LU factorization with partial pivoting on multi-core processors," *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, vol. 0, pp. 353–358, 2012.

[Age10] T. Agerwala, "Exascale computing: the challenges and opportunities in the next decade." in *PPOPP*, R. Govindarajan, D. A. Padua, and M. W. Hall, Eds. J. ACM, 2010, pp. 1–2.

[AHA+11] H. Anzt, V. Heuveline, J. Aliaga, M. Castillo, J. Fernandez, R. Mayo, and E. Quintana-Orti, "Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms," in *Green Computing Conference and Workshops (IGCC), 2011 International*, July 2011, pp. 1–6.

[AHR10] H. Anzt, V. Heuveline, and B. Rocker, "Mixed precision error correction methods for linear systems convergence analysis based on Krylov subspace methods," in *PARA 2010, Part II, LNCS 7134*, K. Jonasson, Ed. Springer, Heidelberg, 2010, pp. 237–248.

[AHR+11a] H. Anzt, V. Heuveline, B. Rocker, M. Castillo, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí, "Power consumption of mixed precision in the iterative solution of sparse linear systems," in *IPDPS Workshops - High-Performance, Power-Aware Computing*, 2011, pp. 829–836.

[AHR11b] H. Anzt, V. Heuveline, and B. Rocker, "An error correction solver for linear systems: Evaluation of mixed precision implementations," in *High Performance Computing for Computational Science – VECPAR 2010*, ser. Lecture

Notes in Computer Science, J. Palma, M. Daydé, O. Marques, and J. Lopes, Eds., vol. 6449. Springer, Berlin / Heidelberg, 2011, pp. 58–70.

[AHW09] W. Augustin, V. Heuveline, and J.-P. Weiss, "Optimized stencil computation using in-place calculation on modern multicore systems," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Springer, 2009, pp. 772–784.

[ALDH12] H. Anzt, P. Luszczek, J. Dongarra, and V. Heuveline, "GPU-accelerated asynchronous error correction for mixed precision iterative refinement," in *Euro-Par*, 2012, pp. 908–919.

[All01] R. Allan, *A History of the Personal Computer: The People and the Technology*. Allan Pub., 2001.

[AMB06] A. T. Are Magnus Bruaset, Ed., *Numerical solution of Partial Differential Equations on Parallel Computers*. Birkhäuser, 2006.

[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 - Research and Development*, vol. 25, no. 3-4, pp. 141–148, 2010.

[ATDH12a] H. Anzt, S. Tomov, J. Dongarra, and V. Heuveline, "A block-asynchronous relaxation method for graphics processing units," in *IPDPS Workshops - Heterogeneous Computing Workshop*, 2012, pp. 113–124.

[ATDH12b] ——, "Weighted block-asynchronous iteration on GPU-accelerated systems," in *Euro-Par Parallel Processing Workshops – HeteroPar*, 2012.

[ATG+12] H. Anzt, S. Tomov, M. Gates, J. Dongarra, and V. Heuveline, "Block-asynchronous multigrid smoothers for GPU-accelerated systems," *Procedia CS*, vol. 9, pp. 7–16, 2012.

[Bag95] R. Bagnara, "A unified proof for the convergence of Jacobi and Gauss-Seidel methods," *SIAM Rev.*, vol. 37, pp. 93–97, March 1995.

[Ban22] S. Banach, "Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales," *Fund. Math.*, vol. 3, pp. 133–181, 1922.

[Bau78] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *J. ACM*, vol. 25, no. 2, pp. 226–244, 1978.

[BB95] T. Burd and R. Brodersen, "Energy efficient CMOS microprocessor design," in *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, 1995.*, vol. 1, January 1995, pp. 288 –297.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[BBC+08] K. Bergman *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," DARPA IPTO ExaScale Computing Study, 2008. [Online]. Available: http://computationalsciencesolutions.com/docs/DARPA%20exascale%20-%20hardware%20(2008).pdf

[BBD+09] M. Baboulin, A. Buttari, J. J. Dongarra, J. Langou, J. Langou, P. Luszczek, J. Kurzak, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526 – 2533, 2009.

[BC10] C. Bekas and A. Curioni, "A new energy aware performance metric," *Computer Science - Research and Development*, vol. 25, pp. 187–195, 2010, 10.1007/s00450-010-0119-z.

[BDB+12] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi," in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science, C. Kaklamanis, T. Papatheodorou, and P. Spirakis, Eds. Springer, Berlin / Heidelberg, 2012, vol. 7484, pp. 477–488.

[BDL+07] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, "Mixed precision iterative refinement techniques for the solution of dense linear systems," *Int. J. of High Performance Computing and Applications*, vol. 21, no. 4, pp. 457–466, 2007.

[BE86] D. P. Bertsekas and J. Eckstein, "Distributed asynchronous relaxation methods for linear network flow problems," *Proceedings of IFAC '87*, 1986.

[BEN88] R. Bru, L. Elsner, and M. Neumann, "Models of parallel chaotic iteration methods," *Linear Algebra and Its Applications*, vol. 103, no. C, pp. 175–192, 1988.

[Ber89] T. J. Bertsekas, D.P., *Parallel and Distributed Computation: Numerical Methods*, 1989.

[Ber05] A. Berger, *Hardware and Computer Organization*, ser. Embedded Technology. Elsevier Science, 2005.

[BFG+] A. H. Baker, R. D. Falgout, T. Gamblin, T. V. Kolev, S. Martin, and U. Meier Yang, "Scaling algebraic multigrid solvers: On the road to exascale," *Proceedings of Competence in High Performance Computing CiHPC 2010*.

[BFKMY11] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. Meier Yang, "Multigrid smoothers for ultra-parallel computing," 2011, lLNL-JRNL-435315.

[BHF+12] P. G. Bridges, M. Hoemmen, K. B. Ferreira, M. A. Heroux, P. Soltero, and R. Brightwell, "Cooperative application/OS DRAM fault recovery," in *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, ser. Euro-Par'11. Berlin / Heidelberg: Springer, 2012, pp. 241–250.

[BMPS99] Z.-Z. Bai, V. Migallón, J. Penadés, and D. B. Szyld, "Block and asynchronous two-stage methods for mildly nonlinear systems," *Numerische Mathematik*, vol. 82, pp. 1–20, 1999.

[BMR97] J. Bahi, J. Miellou, and K. Rhofir, "Asynchronous multisplitting methods for nonlinear fixed point problems," *Numerical Algorithms*, vol. 15, no. 3-4, pp. 315–345, 1997.

[Boj84] A. Bojanczyk, "Optimal asynchronous Newton method for the solution of nonlinear equations." *J. ACM*, vol. 31, no. 4, pp. 792–803, 1984.

[Bou01] M. H. Bourgeois, *Grundlagen der numerischen Mathematik und des wissenschaftlichen Rechnens*. Stuttgart: Teuber, 2001.

[Bra07] D. Braess, *Finite Elemente*. Stuttgart: Springer, 2007.

[BSS99] K. Blathras, D. B. Szyld, and Y. Shi, "Timing models and local stopping criteria for asynchronous iterative algorithms," *Journal of Parallel and Distributed Computing*, vol. 58, pp. 446–465, 1999.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation.* Prentice Hall, 1989.

[BTL10] B. Betkaoui, D. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *International Conference on Field-Programmable Technology (FPT), 2010*, December 2010, pp. 94 –101.

[But08] J. Butcher, *Numerical Methods for Ordinary Differential Equations.* John Wiley and Sons, 2008.

[Cai09] X.-C. Cai, "Nonlinear overlapping domain decomposition methods," in *Domain Decomposition Methods in Science and Engineering XVIII*, ser. Lecture Notes in Computational Science and Engineering. Springer, Berlin / Heidelberg, 2009, vol. 70, pp. 217–224.

[CG10] T. P. Collignon and M. B. Gijzen, "Solving large sparse linear systems efficiently on grid computers using an asynchronous iterative method as a preconditioner," in *Numerical Mathematics and Advanced Applications 2009.* Springer, Berlin / Heidelberg, 2010, pp. 261–268.

[CGG+09] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[CK01] A. T. Chronopoulos and A. B. Kucherov, "A parallel krylov-type method for nonsymmetric linear systems," in *Proceedings of the 8th International Conference on High Performance Computing*, ser. HiPC '01. London, UK, UK: Springer, 2001, pp. 104–114.

[CM69] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and Its Applications*, vol. 2, no. 7, pp. 199–222, 1969.

[CS96] X.-c. Cai and Y. Saad, "Overlapping domain decomposition algorithms for general sparse matrices," Tech. Rep., 1996.

[CSP04] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ser. ISLPED '04. New York, NY, USA: ACM, 2004, pp. 174–179.

[DB91] M. Dubois and F. A. Briggs, "The run-time efficiency of parallel asynchronous algorithms," *IEEE Trans. Computers*, vol. 40, no. 11, pp. 1260–1266, 1991.

[DBB+11] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," Innovative Computing Laboratory, University of Tennessee, Tech. Rep., August 2011.

[DBB+12] ——, "Algorithm-based fault tolerance for dense matrix factorization," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012*, February 2012, pp. 225–234.

[DBM+11] J. Dongarra *et al.*, "The international exascale software project roadmap," *Int. J. of High Performance Computing and Applications*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[Dem97] J. W. Demmel, *Applied Numerical Linear Algebra.* Philadelphia: SIAM, 1997.

[D*et al*11] J. Dongarra *et al*, "The international ExaScale software project roadmap," *Int. J. of High Performance Computing and Applications*, vol. 25, no. 1, 2011.

[DLTD11] P. Du, P. Luszczek, S. Tomov, and J. Dongarra, "Soft error resilient QR factorization for hybrid system with GPGPU," *Journal of Computational Science*, November 2011.

[EB90] D. El Baz, "M-functions and parallel asynchronous algorithms," *SIAM Journal on Numerical Analysis*, vol. 27, no. 1, pp. 136–140, 1990.

[EBFS05] D. El Baz, A. Frommer, and P. Spiteri, "Asynchronous iterations with flexible communication: contracting operators," *J. Comput. Appl. Math.*, vol. 176, no. 1, pp. 91–103, Apr. 2005.

[EBSMG96] D. El Baz, P. Spiteri, J. Miellou, and D. Gazen, "Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems," *Journal of Parallel and Distributed Computing*, vol. 38, no. 1, pp. 1–15, 1996.

[EG93] H. Elman and G. Golub, *Inexact and Preconditioned Uzawa Algorithms for Saddle Point Problems*, ser. Computer science technical report series. University of Maryland, 1993.

[ET82] M. El Tarazi, "Some convergence results for asynchronous algorithms," *Numerische Mathematik*, vol. 39, no. 3, pp. 325–340, 1982.

[Fac00] A. Facius, "Iterative solution of linear systems with improved arithmetic and result verification," Ph.D. dissertation, Universität Karlsruhe, 2000.

[FF12] R. I. Fernandes and G. Fairweather, "An ADI extrapolated Crank-Nicolson orthogonal spline collocation method for nonlinear reaction-diffusion systems," *Journal of Computational Physics*, vol. 231, no. 19, pp. 6248–6267, 2012.

[FL08] Z. Feng and P. Li, "Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 647–654.

[FLP+07] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal, "Analyzing the energy-time trade-off in high-performance computing applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, pp. 835–848, June 2007.

[FoE10] N. S. Foundation and D. of Energy, "BLAS," 2010. [Online]. Available: http://www.netlib.org/blas/

[Fro12] G. Frobenius, *Über Matrizen aus nicht negativen Elementen*, ser. Preussische Akademie der Wissenschaften Berlin: Sitzungsberichte der Preußischen Akademie der Wissenschaften zu Berlin. Reichsdr., 1912.

[Fro94] S. D. Frommer, A., "Asynchronous two-stage iterative methods," *Numer. Math.*, vol. 69, pp. 141–153, 1994.

[Fro98] ——, "Asynchronous iterations with flexible communication for linear systems," *Calculateurs Parallèles*, vol. 10, pp. 421–429, 1998.

[FS00] A. Frommer and D. B. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, pp. 201–216, 2000.

[FSS97] A. Frommer, H. Schwandt, and D. B. Szyld, "Asynchronous weighted additive Schwarz methods," *Electronic Transactions on Numerical Analysis*, vol. 5, pp. 48–61, 1997.

[GC01] J. Goodman and A. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1808 –1820, November 2001.

[GL09] A. Geist and R. Lucas, "Major computer science challenges at exascale," *Int. J. of High Performance Computing and Applications*, vol. 23, no. 4, pp. 427–436, 2009.

[GLGN+08] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *Micro, IEEE*, vol. 28, no. 4, pp. 13–27, 2008.

[GLRE05] L. Giraud, J. Langou, M. Rozložník, and J. v. d. Eshof, "Rounding error analysis of the classical Gram-Schmidt orthogonalization process," *Numerische Mathematik*, vol. 101, pp. 87–100, 2005, 10.1007/s00211-005-0615-4.

[GO87] G. Golub and L. Overton, *The Convergence of Inexact Chebyshev and Richardson Iterative Methods for Solving Linear Systems*, ser. Research report (Australian National University. Centre for Mathematical Analysis), 1987.

[GO11] G. Golub and M. Overton, *Convergence of a Two-Stage Richardson Iterative Procedure for Solving Systems of Linear Equations.* BiblioBazaar, 2011.

[Göd10] D. Göddeke, "Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters," Ph.D. dissertation, Technische Universität Dortmund, Fakultät für Mathematik, May 2010.

[GPT07] C. García, M. Prieto, and F. Tirado, "Multigrid smoothers on multicore architectures," in *PARCO'07*, 2007, pp. 279–286.

[gre] "Green 500 list." [Online]. Available: http://www.green500.org/

[Gre65] D. Greenspan, *Introductory numerical analysis of elliptic boundary value problems*, ser. Harper's series in Modern Mathematics. Harper and Row, 1965.

[Gre87] A. Greenbaum, *Iterative Methods for Solving Linear Systems*, ser. Frontiers in Applied Mathematics. SIAM, 1987.

[GS11] D. Göddeke and R. Strzodka, "Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid," *IEEE Transactions on Parallel and Distributed Systems (TPDS), Special Issue: High Performance Computing with Accelerators*, vol. 22, no. 1, pp. 22–32, 2011.

[GSMy+08] D. Göddeke, R. Strzodka, J. Mohd-yusof, P. Mccormick, H. Wobker, C. Becker, and S. Turek, "Using GPUs to improve multigrid solver performance on a cluster. intnl," *J. of Computational Science and Engineering*, 2008.

[GST07] D. Göddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in fem simulations," *Int. J. of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, 2007.

[GT01] D. Gilbarg and N. Trudinger, *Elliptic Partial Differential Equations of Second Order*, ser. Classics in Mathematics. Springer, 2001.

[GT08] H. Grossauer and P. Thoman, "GPU-based multigrid: Real-time performance in high resolution nonlinear image processing," in *Computer Vision Systems*, ser. Lecture Notes in Computer Science, A. Gasteratos, M. Vincze, and J. Tsotsos, Eds. Springer, Berlin / Heidelberg, 2008, vol. 5008, pp. 141–150, 10.1007/978-3-540-79547-6_14.

[GVL96] G. Golub and C. Van Loan, *Matrix Computations*, ser. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996.

[GY97] G. H. Golub and Q. Ye, "Inexact preconditioned conjugate gradient method with inner-outer iteration," *SIAM J. on Scientific Computing*, vol. 21, pp. 1305–1320, 1997.

[Hac85] W. Hackbusch, *Multigrid Methods and Applications*. Springer, 1985.

[HC03] G. Houzeaux and R. Codina, "An iteration-by-subdomain overlapping Dirichlet/Robin domain decomposition method for advection-diffusion problems," *J. Comput. Appl. Math.*, vol. 158, no. 2, pp. 243–276, Sep. 2003.

[Hen10] N. Henze, *Stochastik für Einsteiger: Eine Einführung in die faszinierende Welt des Zufalls; mit über 220 Übungsaufgaben und Lösungen.* Vieweg+Teubner, 2010.

[HH07] J. Hubbard and B. Hubbard, *Vector Calculus, Linear Algebra, and Differential Forms: A Unified Approach.* Matrix Editions, 2007.

[HH11] M. A. Heroux and M. Hoemmen, "Fault-tolerant iterative methods via selective reliability," Sandia National Laboratories, Tech. Rep., 2011.

[Hig96] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 1996.

[HK12] A. Habermaier and A. Knapp, "On the correctness of the SIMT execution model of GPUs," Fakultät für Angewandte Informatik der Universität Augsburg, Tech. Rep. 2012-01, 2012.

[Hof05] H. Hofstee, "Power efficient processor architecture and the cell processor," in *HPCA-11. 11th International Symposium on High-Performance Computer Architecture, 2005*, February 2005, pp. 258 – 262.

[HS96] V. Heuveline and M. Sadkane, "Chebyshev acceleration techniques for large complex non-Hermitian eigenvalue problems," *Reliable Computing*, vol. 2, no. 2, pp. 111–117, 1996.

[HS97a] ——, "Arnoldi-Faber method for large non-Hermitian eigenvalue problems," *Electronic Transactions on Numerical Analysis*, vol. 5, pp. 62–76, 1997.

[HS97b] ——, "Parallel computation of polynomials with minimal uniform norm and its application to large eigenproblems," *Journal of Computational and Applied Mathematics*, vol. 82, no. 1-2, pp. 185–198, 1997.

[Hup90] B. Huppert, *Angewandte Lineare Algebra*. Walter de Gruyter, 1990.

[HY00] V. E. Henson and U. M. Yang, "BoomerAMG: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, 2000.

[IG12] S. Iqbal and S. Gao. (2012, February) GPUdirect improves communication bandwidth between GPUs on the C410X. Dell TechCenter.

[IGM12] S. Iqbal, S. Gao, and T. Mckercher. (2012, February) Comparing GPU-direct enabled communication patterns for oil and gas simulations. Dell TechCenter.

[int] "Intel C++ compiler options," Intel Corporation, document Number: 307776-002US.

[Kah96] W. Kahan, "Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic," May 1996. [Online]. Available: http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps

[KBD10] J. Kurzak, D. Bader, and J. Dongarra, *Scientific Computing with Multicore and Accelerators*, ser. Chapman and Hall/CRC computational science series. CRC Press, 2010.

[KGMS97] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 184–193.

[Kin88] J. Kingdon, *East African Mammals: An Atlas of Evolution in Africa, Volume 1–7, Part B: Large Mammals*, ser. East African Mammals. University of Chicago Press, 1988.

[KK03] G. Karniadakis and R. Kirby, *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*. Cambridge University Press, 2003.

[KMP11] P. Kornerup, J.-M. Muller, and A. Panhaleux, "Performing arithmetic operations on round-to-nearest representations," *IEEE Transactions on Computers*, vol. 60, pp. 282–291, 2011.

[Kot01] M. Kot, *Elements of Mathematical Ecology*, ser. Elements of Mathematical Ecology. Cambridge University Press, 2001.

[Kul08] U. Kulisch, *Computer arithmetic and validity: theory, implementation, and applications*, ser. De Gruyter studies in Mathematics. Walter De Gruyter, 2008.

[Lam10] F. Lampe, *Green-IT, Virtualisierung und Thin Clients : Mit neuen IT-Technologien Energieeffizienz erreichen, die Umwelt schonen und Kosten sparen*. Vieweg + Teubner, 2010.

[Lan09] W. B. Langdon, "A CUDA SIMT interpreter for genetic programming," Department of Computer Science, King's College London, Strand, WC2R 2LS, UK, Tech. Rep. TR-09-05, 18 Jun. 2009, revised.

[LM86] B. Lubachevsky and D. Mitra, "A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius," *J. ACM*, vol. 33, pp. 130–150, January 1986.

[LRS90] P. J. Lanzkron, D. J. Rose, and D. B. Szyld, "Convergence of nested classical iterative methods for linear systems," *Numerische Mathematik*, vol. 58, pp. 685–702, 1990, 10.1007/BF01385649.

[Luk12] D. Lukarski, "Parallel sparse linear algebra for multi-core and many-core platforms – parallel solvers and preconditioners," Ph.D. dissertation, Karlsruhe Institute of Technology, 2012.

[Mad06] A. Madzvamuse, "Time-stepping schemes for moving grid finite elements applied to reaction–diffusion systems on fixed and growing domains," *Journal of Computational Physics*, vol. 214, no. 1, pp. 239 – 263, 2006.

[MBdD+09] J. Muller, N. Brisebarre, F. de Dinechin, C. Jeannerod, L. Vincent, and G. Melquiond, *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2009.

[McC87] S. McCormick, *Multigrid Methods*, ser. Frontiers in Applied Mathematics. SIAM, 1987.

[Mei05] A. Meister, *Numerik linearer Gleichungssysteme*. Vieweg, 2005.

[Mey00] C. D. Meyer, Ed., *Matrix analysis and applied linear algebra*. Philadelphia, PA, USA: SIAM, 2000.

[MRK10] K. Malkowski, P. Raghavan, and M. Kandemir, "Analyzing the soft error resilience of linear solvers on multicore multiprocessors," *Parallel Distributed Processing (IPDPS)*, pp. 1 –12, April 2010.

[Mur03] J. Murray, *Mathematical Biology II: Spatial Models and Biomedical Applications*, ser. Interdisciplinary Applied Mathematics. Springer, 2003.

[MVM09] F. Miller, A. Vandome, and J. McBrewster, *History of Computing Hardware (1960s - Present), Human Computer, Analog Computer, Colossus Computer, Eniacand Mainframe Computer*. Alphascript Publishing, 2009.

[MY11] U. Meier Yang, "On the use of relaxation parameters in hybrid smoothers," *Numerical Linear Algebra with Applications*, vol. 11, pp. 155–172, 2011.

[NVI] *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, NVIDIA Corporation.

[NVI09] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2nd ed., NVIDIA Corporation, August 2009.

[NVI11] *CUDA Toolkit 4.0 Readiness for CUDA Applications*, 4th ed., NVIDIA Corporation, March 2011.

[NX06] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *Computer*, vol. 39, no. 1, pp. 118 – 120, January 2006.

[OLG+07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," 2007.

[OR70] J. Ortega and W. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, 1970.

[Par80] B. Parlett, *The symmetric eigenvalue problem*, ser. Prentice-Hall series in Computational Mathematics. Prentice-Hall, 1980.

[PBGM09] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *Proceedings of the 36th annual International Symposium on Computer Architecture*, ser. ISCA. New York, NY, USA: ACM, 2009, pp. 93–104.

[PH04] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, ser. The Computer Architecture and Design Series. Elsevier/Morgan Kaufmann, 2004.

[Pot98] M. Pott, "On the convergence of asynchronous iteration methods for nonlinear paracontractions and consistent linear systems," *Linear Algebra and Its Applications*, vol. 283, no. 1-3, pp. 1–33, 1998.

[pow09] *Power-Aware Scheduling of Virtual Machines in DVFS-enabled Clusters*. New Orleans, LA: IEEE Computer Society, August 2009.

[Pro92] G. Prota, *Melanins and Melanogenesis.* Academic Press, 1992.

[QSS00] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical Mathematics*, ser. Texts in Applied Mathematics. Springer, 2000.

[QV99] A. Quarteroni and A. Valli, *Domain Decomposition Methods for Partial Differential Equations*, ser. Numerical Mathematics and Scientific Computation. Oxford University Press, USA, 1999.

[Rab89] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *J. ACM*, vol. 36, no. 2, pp. 335–348, Apr. 1989.

[Ral79] L. Rall, *Computational solution of nonlinear operator equations.* R. E. Krieger, 1979.

[Ran08] R. Rannacher, *Numerische Mathematik 2 – Numerik partieller Differentialgleichungen*, Vorlesungsskriptum, Ed. Institut für Angewandte Mathematik Universität Heidelberg, 2008. [Online]. Available: http://numerik.iwr.uni-heidelberg.de/~lehre/notes/

[RCN04] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital integrated circuits-A design perspective*, 2nd ed. Prentice Hall, 2004.

[RHMDR07] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye, "Towards optimal multi-level tiling for stencil computations," in *Parallel and Distributed Processing Symposium (IPDPS) IEEE International*, March 2007, pp. 1 –10.

[Rus78] R. M. Russell, "The Cray-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[Ruu95] S. J. Ruuth, "Implicit-explicit methods for reaction-diffusion problems in pattern formation," *Journal of Mathematical Biology*, vol. 34, pp. 148–176, 1995, 10.1007/BF00178771.

[Rya11] D. Ryan, *History of Computer Graphics: DLR Associates Series.* AuthorHouse, 2011.

[Saa03] Y. Saad, *Iterative Methods for Sparse Linear Systems.* Philadelphia, PA, USA: SIAM, 2003.

[SBKK98] Y. Su, A. Bhaya, E. Kaszkurewicz, and V. S. Kozyakin, "Further results on convergence of asynchronous linear iterations," *Linear Algebra and its Applications*, vol. 281, no. 1-3, pp. 11–24, 1998.

[Sch79] J. Schnakenberg, "Simple chemical reaction systems with limit cycle behaviour," *Journal of Theoretical Biology*, vol. 81, no. 3, pp. 389 – 400, 1979.

[Sch97] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, Jun. 1997.

[She94] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Pittsburgh, PA, USA, Tech. Rep., 1994.

[SK06] H.-R. Schwarz and N. Köckler, *Numerische Mathematik.* Stuttgart: Teuber, 2006.

[SMB+02] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *Proceedings of eighth International Symposium on High-Performance Computer Architecture*, February 2002, pp. 29 – 40.

[SSH+03] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware computer systems: Opportunities and challenges," *Micro, IEEE*, vol. 23, no. 6, pp. 52 – 61, 2003.

[Str97] J. C. Strikwerda, "A convergence theorem for chaotic asynchronous relaxation," *Linear Algebra and its Applications*, vol. 253, no. 1-3, pp. 15–24, Mar. 1997.

[Stu01] K. Stueben, "A review of algebraic multigrid," *J. Comput. Appl. Math.*, vol. 128, no. 1-2, pp. 281–309, Mar. 2001.

[Sup10] *Supermicro X8DTG-QF User's Manual*, Revision 1.0a ed., Super Micro Computer, Inc., 2010.

[SZ10] M. Steffen and J. Zambreno, "Improving SIMT efficiency of global rendering algorithms with architectural support for dynamic micro-kernels," in *MICRO*, 2010, pp. 237–248.

[Szy98a] D. B. Szyld, "Different models of parallel asynchronous iterations with overlapping blocks," *Computational and Applied Mathematics*, vol. 17, pp. 101–115, 1998.

[Szy98b] ——, "The mystery of asynchronous iterations convergence when the spectral radius is one," Department of Mathematics, Temple University, Philadelphia, Pa., Tech. Rep. 98-102, October 1998. [Online]. Available: http://www.math.temple.edu/~szyld/reports/mystery.pdf

[Tho08] P. Thoman, *Multigrid Methods on GPUs: High Performance GPGPU Solvers for Partial Differential Equations.* VDM Publishing, 2008.

[top] *TOP 500 list.* [Online]. Available: http://www.top500.org/

[Tre02] N. Trefethen, "Hundred-dollar, hundred-digit challenge problems," *SIAM News*, vol. 35, no. 1, January 2, 2002, problem no. 7.

[Tro00] U. Trottenberg, *Multigrid.* Academic Press, 2000.

[ÜD86] A. Üresin and M. Dubois, "Generalized asynchronous iterations," in *CONPAR*, 1986, pp. 272–278.

[ÜD96] ——, "Effects of asynchronism on the convergence rate of iterative algorithms," *J. Parallel Distrib. Comput.*, vol. 34, no. 1, pp. 66–81, 1996.

[Var04] R. Varga, *Geršgorin and his Circles.* Springer, 2004.

[Var10] ——, *Matrix Iterative Analysis*, ser. Springer Series in Computational Mathematics. Springer, 2010.

[Wes92] P. Wesseling, *An introduction to multigrid methods.* John Wiley and Sons Inc, 1992.

[WLB00] S. F. M. William L. Briggs, Van Emden Henson, *A Multigrid Tutorial.* SIAM, 2000.

[Xu99] J.-J. Xu, "Convergence of partially asynchronous block quasi-Newton methods for nonlinear systems of equations," *Journal of Computational and Applied Mathematics*, vol. 103, no. 2, pp. 307–321, 1999.

[YHE02] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," in *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM). Proceedings. IEEE*, vol. 3, 2002, pp. 1567 – 1576 vol.3.