Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms

Parallel Solvers and Preconditioners

Zur Erlangung des akademischen Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

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

DISSERTATION

von
M.Sc. Dimitar Lukarski
aus
Sofia

Tag der mündlichen Prüfung: 25.01.2012

Referent: Junior-Prof. Dr. Jan-Philipp Weiß
Karlsruher Institut für Technologie, Deutschland
Korreferent: Prof. Dr. Vincent Heuveline
Karlsruher Institut für Technologie, Deutschland
Korreferent: Assistant Prof. Richard Vuduc, Ph.D.
Georgia Institute of Technology, Atlanta, USA

Abstract

Partial differential equations are typically solved by means of finite difference, finite volume or finite element methods resulting in large, highly coupled, ill-conditioned and sparse (non-)linear systems. In order to minimize the computing time we want to exploit the capabilities of modern parallel architectures. The rapid hardware shifts from single core to multi-core and many-core processors lead to a gap in the progression of algorithms and programming environments for these platforms – the parallel models for large clusters do not fully utilize the performance capability of the multi-core CPUs and especially of the GPUs. Software stack needs to run adequately on the next generation of computing devices in order to exploit the potential of these new systems. Moving numerical software from one platform to another becomes an important task since every parallel device has its own programming model and language. The greatest challenge is to provide new techniques for solving (non-)linear systems that combine scalability, portability, fine-grained parallelism and flexibility across the assortment of parallel platforms and programming models. The goal of this thesis is to provide new fine-grained parallel algorithms embedded in advanced sparse linear algebra solvers and preconditioners on the emerging multi-core and many-core technologies.

With respect to the mathematical methods, we focus on efficient iterative linear solvers. Here, we consider two types of solvers – out-of-the-box solvers such as preconditioned Krylov subspace solvers (e.g. CG, BiCGStab, GMRES), and problem-aware solvers such as geometric matrix-based multi-grid methods. Clearly, the majority of the solvers can be written in terms of sparse matrix-vector and vector-vector operations which can be performed in parallel. Our aim is to provide parallel, generic and portable preconditioners which are suitable for multi-core and many-core devices. We focus on additive (e.g. Gauss-Seidel, SOR), multiplicative (ILU factorization with or without fill-ins) and approximate inverse preconditioners. The preconditioners can also be used as smoothing schemes in the multi-grid methods via a preconditioned defect correction step. We treat the additive splitting schemes by a multi-coloring technique to provide the necessary level of parallelism. For controlling the fill-in entries for the ILU factorization we propose a novel method which we call the *power(q)-pattern method*. We prove that this algorithm produces a new matrix structure with diagonal blocks containing only diagonal entries. This approach provides higher degrees of parallelism in comparison with the level-scheduling/topological sort algorithm. With these techniques we can perform the forward and backward substitution of the preconditioning step in parallel. By formulating the algorithm in block-matrix form we can execute the sweeps in parallel only by performing matrix-vector multiplications. Thus, we can express the data-parallelism in the sweeps without any specification of the underlying hardware or programming models.

In object-oriented languages, an abstraction separates the object behavior from its implementation. Based on this abstraction, we have developed a linear algebra toolbox which supports several platforms such as multi-core CPUs, GPUs and accelerators. The various backends (sequential, OpenMP, CUDA, OpenCL) consist of optimized and platform-specific matrix and vector routines. Using unified interfaces across all platforms, the library allows users to build linear solvers and preconditioners without any information about the underlying hardware. With this technique, we can write our solvers and preconditioners in a single source code for all platforms. Furthermore, we can extend the library by adding new platforms without modifying the existing solvers and preconditioners.

In our tests we consider two scenarios – preconditioned Krylov subspace methods and matrix-based multi-grid methods. We demonstrate speed ups in two directions: first, the preconditioners/smoothers reduce the total solution time by decreasing the number of iterations, and second, the preconditioning/smoothing phase is efficiently executed in parallel providing good scalability across several parallel architectures. We present numerical experiments and performance analysis on several platforms such as multi-core CPU and GPU devices. Furthermore, we show the viability and benefit of the proposed preconditioning schemes and software approach.

Acknowledgements

This thesis would not have been possible without my adviser, Jan-Philipp Weiss, who has been guiding me since my first years in Karlsruhe and introducing me in the field of parallel and high performance scientific computing. I would also like to give a special acknowledgment to Vincent Heuveline for his inspiring and constructive discussions throughout my work.

Furthermore, I would like to thank all of my colleagues and friends in the Engineering Mathematics and Computing Lab at Karlsruhe Institute of Technology for their continuous support and encouragement. I would like to express my gratitude to Felix Riehn, Nico Trost and Niels Wegh for their contributions to parts of our software stack.

I am also grateful to the Karlsruhe Institute of Technology for funding this program in the framework of the German Excellence Initiative and to our collaborating partner Hewlett-Packard.

Last but not least, I would like to express my gratitude to my girlfriend Silvana for her love and encouragement during my work. Furthermore, I want to thank my parents, Anna and Hristo, for their support during my odyssey in the research world.

Contents

List of Figures

List of Tables

Nomenclature

BiCGStab	–	Bi-Conjugate Gradient Stabilized method
BJ	–	Block Jacobi
BLAS	–	Basic Linear Algebra Subprograms
BLAS1	–	BLAS routines level 1 - vector-vector routines
BLAS2	–	BLAS routines level 3 - matrix-vector routines
BLAS3	–	BLAS routines level 2 - matrix-matrix routines
CG	–	Conjugate Gradient method
CPU	–	Central Processing Unit
CSR	–	Compress Sparse Row matrix format
CUDA	–	Compute Unified Device Architecture
DMA	–	Direct Memory Access
DOF	–	Degree Of Freedom(s)
FE	–	Finite Element
FEM	–	Finite Element Method
Flop	–	Floating-point operation(s)
FSAI	–	Factorized SpArse Inverse
GMRES	–	Generalized Minimal RESidual method
GPU	–	Graphics Processing Unit
GS	–	Gauss-Seidel scheme
HPC	–	High Performance Computing
I/O	–	Input/Output
ILU	–	Incomplete LU factorization
ILU(0)	–	Incomplete LU factorization with no fill-ins
ILU(p)	–	Incomplete LU factorization with fill-ins based on p-levels
ILU(p,q)	–	power(q)-pattern method with ILU(p) factorization
LAToolbox	–	Linear Algebra Toolbox
lmpLAToolbox	–	local multi-platform LAToolbox
MPI	–	Message Passing Interface
$\mathcal{NNZ}(A)$	–	Non-zero pattern of the matrix A
OpenMP	–	Open Multi-Processing
PDE	–	Partial Differential Equation
RAM	–	Random Access Memory
SGS	–	Symmetric Gauss-Seidel scheme
SOR	–	Successive Over-Relaxation scheme
SpMV	–	Sparse Matrix-Vector multiplication
SSOR	–	Symmetric Successive Over-Relaxation scheme

Chapter 1

Introduction

1.1 Motivation

1.1.1 Scientific Computing – Simulations and Efficient Numerical Methods

Many physical phenomena can be described by Partial Differential Equations (PDEs). Typically, these equations can be solved efficiently by means of finite difference/element/volume methods [29, 37, 54, 76, 97]. The finite approximations of these problems result in large and sparse systems. For many applications, the iterative methods have been proven to be one of the most efficient solution schemes. For various sparse linear problems, the preconditioned Krylov subspace solvers (e.g. preconditioned CG, BiCGStab and GMRES) provide low complexity in comparison to other iterative solvers such as splitting methods [15, 42, 114]. In addition, they can be used as out-of-the-box solvers without any information about the underlying linear system. On the other hand, multi-grid methods relying on hierarchical grid decomposition techniques give linear complexity with respect to the number of unknowns in the system and thus, for many elliptic problems, they are one of the most efficient and optimal solvers [117, 124].

Scientific computing is a field of study which focuses on numerical methods and their realization. For linear solvers, the performance of the solution process depends on the characteristics of the used method and its realization on a particular computer system [48, 114]. Typically, minimizing the run time and increasing the performance depend on the complexity and the efficiency of the selected method, algorithmic optimization and hardware-specific tuning, see Figure 1.1. Especially in the area of sparse linear solvers, the choice of a proper and efficient method, and a

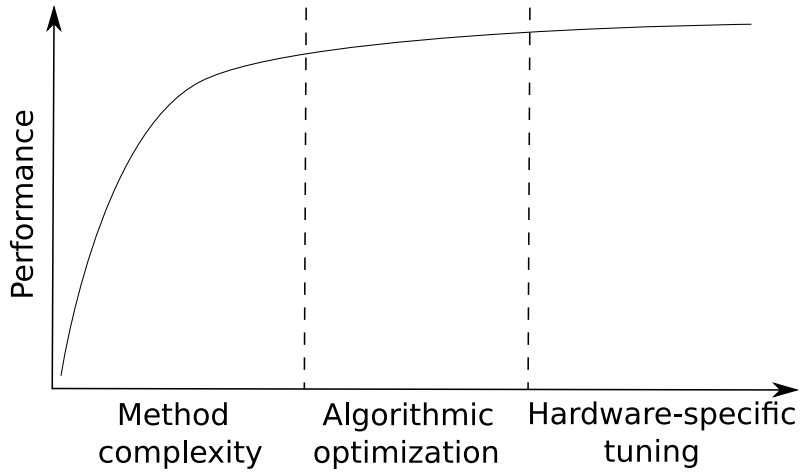

Figure 1.1: Abstraction of typical performance gains in a scientific computation application

solution scheme with good characteristics (e.g. memory storage, degree of parallelism, etc) has

the largest impact on the performance profile. For many applications, algorithm optimization and better software design can deliver additional improvement and performance gain. Typically, hardware-specific tuning such as loop unrolling and peeling, instruction-level-parallelism, vector units, etc can speed up the solution phase only by a limited factor.

Furthermore, the time of the solution process depends on the specific hardware features of the system. Therefore, we need to ensure that the proper solution process takes into account the characteristics of the selected method, specific implementation details and hardware features. Numerical methods that are aware of the hardware features and utilize the platform efficiently provide the best performance.

1.1.2 Hardware Shifts

For most of the software products, single core processors have been proven to provide good hardware performance, portability and forward compatibility. In this context, portability and compatibility are defined as the ability to move a program from one computer system to another without any code modification. To increase the performance of the single core processors, the major micro-processor producers rely strongly on hardware improvements such as instruction pipe-lining, out-of-order execution, pre-fetching schemes and, most important, increase of the clock frequency [56]. These techniques ensure better performance on the majority of sequential programs. However, due to physical limitations of the semiconductor technology these trends are not sustainable [9]. One of the major obstacles in continuing to increase the clock frequency is the power constraint combined with heat dissipation restrictions. In the last few years the combined restrictions of the memory bandwidth and latency as well as the limited acceleration factors of the instruction level parallelism have caused a hardware shift – moving from single-core to multi-core and many-core processors and devices.

1.1.3 Emerging Multi-core and Many-core Devices

New emerging multi-core and many-core technologies mostly differ from the previous single-core concept by providing more cores on the chip. Furthermore, the internal memory structure of the micro-processors is evolving – the local internal processor memory is moving from caches that are large, automatic and transparent to small and mostly manually managed local or shared memory. This is a necessary step in order to provide a scalable internal memory system for handling the accesses and transfers from the global memory to the processor. In addition, the compute power is rearranged from a few fat computational cores to many lighter compute units in different homogeneous or heterogeneous setups. Typical examples are Graphics Processing Unit (GPU) devices [104, 105], Sony Toshiba IBM Cell Broadband Engine (STI Cell BE) processor [67] and state-of-the-art technologies such as Intel Many Integrated Core (MIC) or Single-Chip Cloud (SCC) architecture [72, 74].

1.1.4 Software Impact

The hardware shifts and emerging multi-core and many-core devices cause a significant software impact. 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 peak performance gap between single core and multi-core/many-core devices, the single-threaded programs tend to perform even worse on the emerging platforms – theoretically, on a dual core system (typical Intel/AMD CPUs in 2006 [71]) a sequential program would utilize 50% of the peak performance of the machine, while on a 500-core chip (typical NVIDIA GPUs in 2011 [104]) it would utilize only 0.2%. Furthermore, programs designed for clusters do not utilize the full power potential of modern multi-core CPUs due to the different synchronization mechanisms. These programs are not able to run on any of the GPU devices, since none of the many-core platforms support explicit communication control.

Thus, especially in the scientific computing, many of the programs have to be re-programmed and re-designed in order to achieve full utilization of these new systems.

1.1.5 Programming Models and Languages

The new field of many-core processors imposes a shift in various software concepts and programming paradigms. Without clear vision about the future architecture with respect to the memory managements and core communication mechanisms, we observe a large variety of programming models such as fork-and-join, point-to-point communication, global address space and data-stream model. This results in programming languages like OpenMP [106], IBM Cell SDK [66], UPC [22], CUDA [100], OpenCL [100], Cilk plus [70], Thread Building Blocks [75] and many more.

Many numerical algorithms and schemes have been developed in sequential manner. This implies not only a necessity of re-programming but also re-designing in order to move the programs to this new era of parallel devices. Many of the numerical schemes and algorithms need to be adapted to this new technology. In some cases, a parallel algorithm could have higher complexity and more sophisticated steps than its sequential pre-successor. In addition, the parallelism of an algorithm could imply also a scalable and efficient implementation only on a particular hardware platform. This, of course, leads to limited portability and flexibility of the new software. Thus, it becomes an important issue to provide a sustainable efficient parallel programs which are portable across different hardware platforms.

1.1.6 Numerical Adaptation and Challenges

The adaptation of many numerical methods for multi-core and many-core devices becomes a critical aspect for modern simulations. In the recent years several research groups have been working on developing solutions to this problem. The process has started with migrating and adapting some of the numerical methods to the new parallel devices. However, this leads to isolated solutions for many applications and devices such as CG solver on IBM Cell BE [60], lattice Boltzmann methods on ClearSpeed boards [63] and others.

Next we need to develop portable numerical methods that are not restricted to specific devices. This interdisciplinary task not only requires an adaptation of the numerical schemes with respect to the fine-grained parallelism but it also requires a generic strategy to maintain an abstraction of the underlying programming model. The critical goal is to achieve hardware and software abstraction.

Currently, several linear algebra libraries are under development. General sparse linear algebra routines are provided in libraries as ViennaCL [112] and LAMA [44]. However, they provide limited functionality, especially with respect to the preconditioning schemes which are important for many applications. A geometric multi-grid solver supporting several platforms has been developed in the FEAST project [48]. Built on a locally structured grid, the smoothers are developed to work only with structured matrices.

General numerical methods and libraries for sparse linear systems suited for multi-core and many-core platform are still missing. The aim of this work is to provide the next step in this field of research in terms of new numerical algorithms with fine-grained parallelism.

1.2 Goal and Thesis Contributions

Implementation of efficient and scalable numerical methods on highly-parallel processors is an interdisciplinary task which requires knowledge in applied mathematics and computer science. Our aim is to provide new highly parallel schemes and a generic software framework for building different iterative solvers on multi-core and many-core platforms. Special focus goes to the preconditioning equation, where we propose a new technique for the incomplete LU-factorization

with fill-ins. Furthermore, we rely on the block-wise form of the solution scheme and therefore bring a high level of abstraction for additive (splitting), multiplicative and approximate inverse preconditioners. This abstraction leads to portable solvers that cover various of programming models and hardware specifications.

The contributions of this thesis are summarized as follows:

Fine-grained Parallel Preconditioners To provide a fine-grained level of parallelism we propose a block-wise solution scheme form of the LU-based preconditioners for additive and multiplicative schemes. Since the sparsity pattern of the preconditioning matrix is not changed, the additive (splitting) preconditioners are treated with multi-coloring decomposition. For the incomplete LU-factorization with fill-ins we propose a new method, called *power(q)-pattern method*, in which we control the sparsity pattern. Thus, we can construct the preconditioning matrix in a special way to provide high degree of parallelism for the forward and backward substitutions.

Multi-platform Library Design From an implementation standpoint, we develop a multi-platform linear algebra toolbox which supports several backends such as multi-core CPUs, GPUs and accelerators. Based on abstract data types and unified interfaces for all platforms, we provide a new library to build a numerical scheme without any hardware-specific information. We implement several iterative methods and preconditioners which can be executed efficiently in parallel on different backend platforms.

Numerical Examples and Benchmarks Analysis To quantify the proposed parallel strategy and software library we consider two types of solvers – preconditioned Krylov subspace solvers (CG, BiCGStab, GMRES) and a matrix-based multi-grid method. We focus on the quality of the preconditioners/smoothers by showing acceleration factors in terms of reduction of the number of iterations. Furthermore, we present speed ups and benchmark profiles on various systems, show the computing time with respect to different preconditioners/smoothers and display the benefits of using multi-core and many-core systems. We demonstrate several benchmarks based on different hardware configurations (CPUs and GPUs).

1.3 Thesis Outline

In Chapter 2 we provide a brief introduction to the finite elements method for elliptic PDEs. We continue with an overview of the linear solvers and focus on iterative methods which include splitting methods, Krylov subspace methods and multi-grid methods. Furthermore, we emphasize the importance of the preconditioning phase and we demonstrate some of the most common techniques for building the preconditioning equation. We conclude the chapter with remarks on the algorithmic complexity and computational intensity.

In Chapter 3 we present parallel techniques for performing linear algebra, solvers and preconditioners. We start with the parallelism of the basic linear algebra routines (BLAS) including some comments and remarks on accuracy and consistency of the results. We continue with the parallel solvers – presenting our concept for full-parallel and hybrid-parallel schemes. Special focus goes to describing parallel execution of preconditioners based on additive and multiplicative decomposition. By formulating the forward and backward substitutions in a block-matrix form we can execute them in parallel by only performing matrix-vector multiplications without any specification of the underlying hardware or programming models. We treat the additive splitting schemes by a multi-coloring technique to provide the necessary level of parallelism. For controlling the fill-in entries in the incomplete LU factorization we propose a novel method the *power(q)-pattern method*. This algorithm produces a new matrix structure with diagonal blocks containing only diagonal entries. With this approach we obtain a higher degree of parallelism in

comparison with the level-scheduling/topological sort algorithm. At the end of the chapter we present remarks on other parallel preconditioning schemes.

Our concept and realization of the multi-platform linear algebra toolbox for sparse iterative solvers is presented in Chapter 4. We give an overview of the memory access model and the key object methods which help us to build efficient preconditioners and solvers for various platforms. We also describe the design that allows us to maintain a single source code program.

In Chapter 5 we give a qualitative performance analysis of the parallel linear solvers and preconditioners on a variety of problems. First, we consider the Poisson and the convection-diffusion problems. For the Poisson problem we present performance results for preconditioned CG method and multi-grid solver on a 2D L-shaped locally refined grid. The convection-diffusion equation is solved with preconditioned GMRES. We present results that demonstrate the impact of the physical dimensions (2D, 3D) and of the finite elements (linear, quadratic) on the multi-coloring decomposition. Furthermore, after factorization, we investigate the sparsity patterns for the level-scheduling and power(q)-pattern method. We show the behavior of the preconditioned solvers executed on different matrices. These tests show the strength of the power(q)-pattern method and the ability to use it as an out-of-the-box scheme. For the preconditioned solvers we present the impact on the number of iterations as well as on the total computational time on multi-core CPU and GPU devices.

Chapter 2

Mathematical Background

In this chapter, we present a brief introduction to the finite element methods for elliptic partial differential equations. We continue with an overview of the linear solvers and we focus on iterative methods which include splitting methods, Krylov subspace methods and multi-grid methods. Furthermore, we emphasize the importance of the preconditioning phase and we demonstrate some of the most common techniques for building the preconditioning equation. We conclude this chapter with remarks on the algorithmic complexity and computational intensity.

2.1 Finite Element Methods

The major part of physical phenomena can be modeled by PDEs. Due to the nature of these problems, an analytic solution in most of these cases is not possible to be found. Therefore, the only way to find an approximation to the solution is to solve the problem by using numerical methods. The main aspects of these methods are the efficient solution procedure and control of the errors. Finite Element Methods (FEM) combined with adequate linear solvers provide these key features.

Let us consider a general form of a linear PDE with homogeneous Dirichlet boundary conditions of the form

$$\begin{aligned} Lu &= g \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{2.1}$$

where $\Omega \subset \mathbb{R}^n$ is a bounded domain with sufficiently smooth boundary $\partial\Omega$ ($\partial\Omega = \bar{\Omega} \setminus \Omega$). The differential operator L is defined by

$$L = \sum_{|\beta|,|\gamma| \leq m} (-1)^{|\gamma|} D^{\gamma} a_{\beta\gamma}(x) D^{\beta}$$

and is assumed to be a uniformly elliptic operator in Ω, i.e.

$$\sum_{|\beta|=|\gamma|=m} \xi^{\beta} a_{\beta\gamma}(x) \xi^{\gamma} \geq c|\xi|^{2m} \quad \text{for almost all } x \in \Omega, \xi \in \mathbb{R}^n. \tag{2.2}$$

Here $a_{\beta\gamma}(x) \in L^{\infty}(\bar{\Omega})$, D^{γ} is the partial derivative with multi-index γ which takes values in $\mathbb{Z}^n$ and $|\gamma| = \sum_{i=1}^{n} \gamma_i$. The classical solution of (2.1), when it exists, belongs to $C^{2m}(\Omega) \cap H_0^m(\Omega)$, where C^{2m} are all $2m$ times differentiable functions and

$$H_0^m(\Omega) := \{u \in L^2(K) : D^{\alpha} u \in L^2(K) \text{ for } |\alpha| \leq m \text{ and all } K \subset \Omega \text{ compact}\}.$$

We test the equation with an arbitrary infinitely many times differentiable function with

compact support $v \in C_0^\infty(\Omega)$ and we get

$$(Lu, v) = \sum_{|\beta|,|\gamma|\leq m} (-1)^{|\gamma|} \int_\Omega D^\gamma (a_{\beta\gamma}(x) D^\beta u) v dx.$$

In this way, we obtain a weak solution only in $H_0^m(\Omega)$. Moreover, after integrating by parts, the boundary terms vanish due to the compact support of v (see [54]), and the problem in its weak form reads

$$a(u, v) = f(v) \tag{2.3}$$

for

$$a(u, v) := \sum_{|\beta|,|\gamma|\leq m} \int_\Omega a_{\beta\gamma}(x)(D^\beta u)(D^\gamma v) dx, \tag{2.4}$$

$$f(v) := \int_\Omega g(x) v(x) dx. \tag{2.5}$$

Obviously, a classical solution is also a weak solution. We can present the problem in the following form

$$\text{Find } u \in H_0^m(\Omega) \text{ such that } a(u, v) = f(v), \text{ for all } v \in C_0^\infty(\Omega).$$

Since the space $C_0^\infty(\Omega)$ is dense in $H_0^m(\Omega)$, the weak formulation is equivalent to

$$\text{Find } u \in V \text{ such that } a(u, v) = f(v), \text{ for all } v \in V, \tag{2.6}$$

where V is $H_0^m(\Omega)$, see [54]. Clearly, $a : V \times V \to \mathbb{R}$ is a bounded bilinear form, i.e.

$$|a(u, v)| \leq \varepsilon ||u||_V ||v||_V. \tag{2.7}$$

Furthermore, using (2.2), we can show that a is coercive, i.e. there exists a constant $\alpha > 0$, for all $v \in V$ such that

$$a(v, v) \geq \alpha ||v||_V^2. \tag{2.8}$$

Then, by the Lax-Milgram Theorem, it follows that there exist a unique solution $u \in V$ of problem (2.3) [30].

The main idea behind the Ritz-Galerkin method is to replace the infinite dimensional vector space V by a finite-dimensional space V_h, where $V_h \subset V$ ($\dim V_h < \infty$ and $\dim V = \infty$). The solution of this problem is called Ritz-Galerkin solution with the associated space V_h.

In finite dimensional spaces we can represent the solution $u_h \in V_h$ as a finite linear combination of the basis elements b_i of V_h, namely

$$V_h = \text{span}\{b_1, ..., b_N\},$$
$$u_h = \sum_{i=1}^{N} u_i b_i,$$

where $u_i \in \mathbb{R}$ are the component coefficients of the discrete vector $u_h := (u_1, ..., u_N)$ [54]. Doing so, we can obtain the Ritz-Galerkin solution by solving the linear system

$$Au_h = f_h, \tag{2.9}$$

where $A = (a_{i,j})$ and f_h are defined as

$$a_{i,j} := a(b_i, b_j) \qquad \text{for all } i,j \in \{1,...,N\}, \tag{2.10}$$
$$f_h := (f_i) = f(b_i) \quad \text{for all } i \in \{1,...,N\}. \tag{2.11}$$

One of the key points of a good numerical scheme is the control of the errors in the method. For FEM, the theorem of Céa provides the error estimation

$$||u - u_h||_V \leq \frac{\alpha}{\varepsilon} \inf_{v_h \in V_h} ||u - v_h||_V,$$

where u is the solution of (2.6), u_h is the Ritz-Galerkin solution, α comes from the coercivity condition (2.8) and ε comes from the boundedness condition (2.7) of the bilinear form. In other words, by taking the space V_h closer to the unknown function space V we obtain a better discretization error [30, 43, 54].

The general form of the Ritz-Galerkin method leads to a dense matrix A. This can be avoided by introducing a compact support of the basis functions b_i over some interior section of the original domain Ω. Let us consider a triangulation $T_h = \{T_1, ..., T_M\}$ of our domain Ω into M small sub-domains (called cells) with $\cup_{i=1..M} T_i = \Omega$. The finite elements are local to each cell from the triangulation. This ensures the condition $a_{i,j} = a(b_i, b_j) = 0$ if T_k and T_l have empty intersection for $k, l \in \{1, ..., M\}$, where the i-th node belongs to the cell T_k and the j-th node to the cell T_l. Thus, the integrals (2.4) and (2.5) over the domain Ω can be split into sums of integrals over the triangulation T_h. Then the assembling of the matrix A and the right-hand side vector f_h can be done locally by summing the contributing coefficients.

We can define different finite elements with respect to the different nodal bases of the associate polynomials. Considering only the vertices of a triangular or a rectangular element we can define the Lagrangian linear and bilinear finite elements in 2D domain as follows

$$P_1(K) = a_0 + a_1 x_1 + a_2 x_2, \text{ with } (x_1, x_2) \in K \text{ and } a_i \in \mathbb{R},$$
$$Q_1(K) = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2, \text{ with } (x_1, x_2) \in K \text{ and } a_i \in \mathbb{R}.$$

In this way we can define our space as

$$V_h = \{v \in C^0(\bar{\Omega}) : v|_K \in P_1(K) \text{ for all } K \in T_h\} \tag{2.12}$$

for triangular elements, and as

$$V_h = \{v \in C^0(\bar{\Omega}) : v|_K \in Q_1(K) \text{ for all } K \in T_h\} \tag{2.13}$$

for rectangular elements.

Detailed information about constructing different finite elements and their characteristics can be found in [54, 76].

2.1.1 FEM Solution Procedure

In the general case, we want to solve a time-dependent and (non-)linear PDE. With respect to the time variable, the problem is tackled by an explicit or an implicit numerical integration scheme such as Euler, Runge-Kutta or another multi-step method [31, 97, 122]. Using this technique as well as other, like operator splitting methods, we can solve general PDEs with FEM [46, 97]. A typical FEM solution procedure is presented in Figure 2.1. As input the solver reads information concerning the physical domain and the object geometry. In addition, it reads what kind of boundary function is to be prescribed at a certain geometric section. These data are then passed to the meshing module and after triangulation the initial mesh is produced. The next module is responsible for the definition of Finite Elements (FE) over the mesh and the degrees of freedom

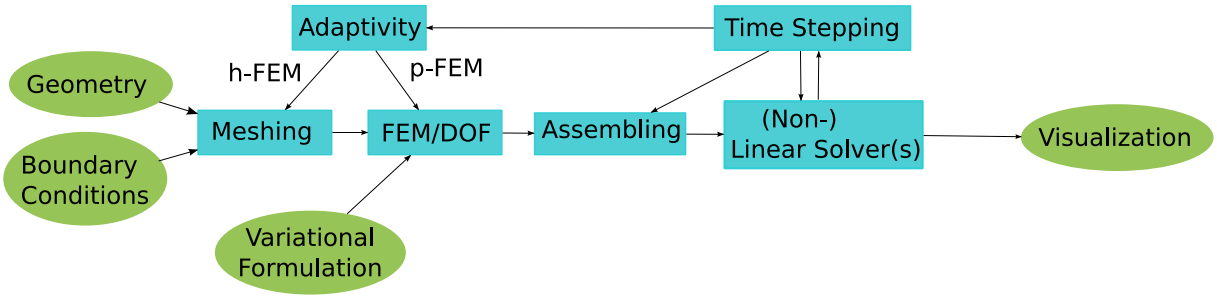

Figure 2.1: FEM solution scheme

over the whole grid. In order to obtain the final matrix (or matrices) and the corresponding right-hand side we need to integrate the discrete formulation (2.10) and (2.11). This integration is usually done on a local level – the integral is computed locally for each cell element of the mesh and the contributing weights are added in the matrix. At this stage we need to solve a linear or non-linear system of equations. The non-linear systems are usually solved with a Newton-like method, where in each iteration we need to compute the inverse of the Jacobian matrix which is a very costly procedure. This can be circumvented by solving a linear system on each step. Thus, we need to solve many linear systems. We can apply several algorithms which are presented further in this section. However, this is the most time consuming operation from the whole solution procedure and therefore it is important to utilize all the available hardware resources. In case of time-dependent model, after the solution of the (non-)linear system has been obtained we need to perform time stepping scheme where another linear system has to be solved. In case of an irregular solution (e.g. grid points on which the solution has a very steep gradient) we can improve the accuracy and the stability of the scheme – based on a local a-posteriori estimation we can select certain (or all) cells of our mesh and then we can either increase the resolution of the spatial disrectization (h-FEM) or we can increase the local polynomial degree of the FE (p-FEM) [40, 43]. Then we need to assemble and solve the new system. At the end we can visualize our results and do post-processing analysis or measurements. Details on the finite element solution procedure can be found in several works [11, 29, 30, 37, 43, 51, 76, 80, 108].

Even in the simplest simulation scenario of a linear and stationary PDE without singular points we need to solve a linear system. Having more complicated models with time stepping and adaptive schemes we need to solve more and larger sparse systems. Therefore, it is of great importance to build not only solvers with low complexity but to minimize the computing time by creating an efficient implementation of the solution process.

2.2 Discrete Representation of Differential Operators

In general, the discrete differential operator A in (2.9) can be represented in two different ways depending of the topology of the grid and the FEs.

2.2.1 Stencils

For rectangular domains in any dimensions (e.g. 2D, 3D) with equidistantly distributed degrees of freedom and constant discretization scheme we can define the discrete differential operator as a stencil. We have a constant relation among all nodes in the domain, details can be found in [51, 118]. This allows us to apply the discrete operator by performing N-dimensional loops with a certain connectivity rule. By using finite difference schemes or lower order FEs this connectivity is a simple weighted combination of the neighboring nodes. An example is presented in Figure 2.2. Here, we consider a 2D square domain with a lexicographical order of the degrees of freedom.

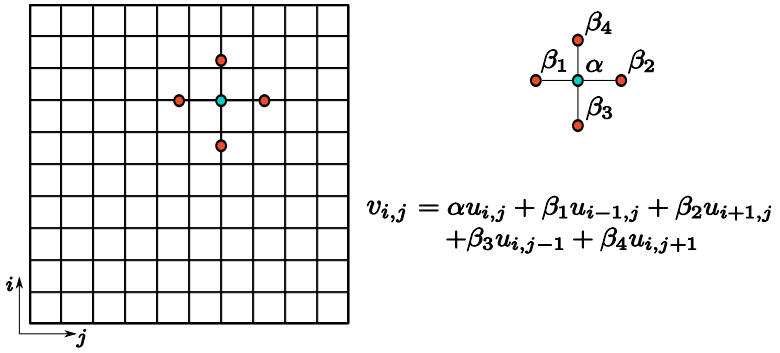

Figure 2.2: Example of a structured grid

The discrete vectors $v_{i,j}$ and $u_{i,j}$ are defined on all grid points $i, j = 1, \ldots, N$. To apply the stencil operator to u and obtain values of v we need to take a linear combination of the given stencil coefficients on every grid point. In the example we present a 5-points stencil – this is a typical discretization scheme based on finite difference methods for the 2D Laplace operator.

The stencil operator, however, imposes several drawbacks when handling more complicated domains and boundary conditions. In this form it does not allow free distribution of the grid points which is essential in cases of singularity or complex geometry.

2.2.2 Sparse Matrices

Independently of the geometry, boundary condition, type of FEs and distribution of the degrees of freedom, we can represent the differential operator by a matrix. The relations between the neighboring points can be written in a matrix form, where after a enumeration, the rows correspond to the node of the system and the columns represent the contributing coefficients from the neighboring points. First, we need to give a unique enumeration to every grid point. The discrete operator on the i-th node of the grid is a linear combination of all the neighboring nodes weighted with the values of the column entries. In Figure 2.3 we present an example of an unstructured grid. By U we denote the set of all nodes for which elements in the triangulation T_h contain the i-th node. In this example, the γ-weight contributing coefficients are written in

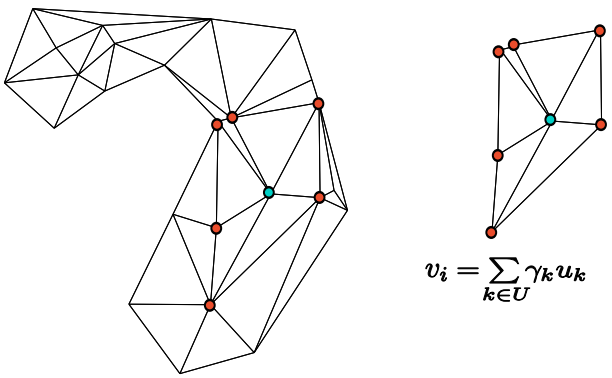

Figure 2.3: Example of an unstructured grid

the i-column of the matrix A. To apply the operator A to u and to obtain the resulting value v we can apply a sparse matrix-vector multiplication. As we mentioned, the relation between two nodes i and j is described in the matrix coefficient $a_{i,j}$ (see (2.10)). Due to the fact that our nodal-basis functions have locally compact support, the stiffness matrix A contains a large amount of zero elements, i.e. it is a sparse matrix, see [29, 76, 108, 118]. We denote the non-zero

matrix pattern (sparsity pattern) of an N-by-N matrix $A = (a_{i,j})_{i,j=1,\ldots,N}$ by

$$\mathcal{NNZ}(A) := \{(i,j) \mid a_{i,j} \neq 0, \quad i,j = 1,\ldots,N\}.$$

The sparsity structure of the matrix depends on the differential operator (e.g. first order, second order, etc), the physical space dimension of the problem (e.g. 2D, 3D), the discretization scheme (e.g. finite differences/elements/volumes), the grid distribution and the enumeration. The matrix structure can also be represented as a directed graph where the grid points (the rows in the matrix) define the nodes, the non-zero column elements represent the vertices, and their values determine the weights. Thus, we can define the connectivity of a graph as a collection of all vertices.

For coercive elliptic problems the diagonal elements are non-zeroes. By using Ritz-Galerkin method (i.e. using the same space for the test and the trial functions) we obtain $a_{i,i} := a(b_i, b_i) > 0$ for all i from $1,\ldots,N$, see (2.8). The resulting matrix is sparse. There are several formats to store such matrices effectively, details can be found in [15, 114].

2.3 Linear Solvers

The linear solvers mainly fall into two categories - direct and iterative methods. The complexity of the direct solvers depends on the matrix size and the connections in the graph of the sparse matrix A. The solution of the system can be obtained only after performing a pre-defined number of steps. This is in contrast to the iterative solvers where an approximation to the solution can be extracted at each iteration step of the solving procedure. The efficiency (i.e. convergence properties) of these schemes typically depends on the condition number and the spectral distribution of the eigenvalues of the matrix. From all of the above mentioned methods, the multi-grid methods have the lowest complexity but they require additional information of the system. Characteristics with respect to convergence properties of the linear solvers can be found in [35, 41, 42, 50, 114, 124].

Direct solvers typically perform a Gaussian elimination (or variation of it) to obtain the solution of the system [41]. For sparse matrices the computational complexity can be decreased from $O(N^3)$ to lower, where N is the dimension of the linear system. Parallelism can be obtained by performing a multi-frontal method instead of the classical Gaussian elimination [2, 3]. However, during the elimination process the sparsity structure of the matrix is lost. In this case, a memory-saving strategy can be introduced by only temporary storing the fill-ins. For small matrices with low connectivity these techniques are very efficient and they provide a fast solution process. Unfortunately, for very large linear systems arising from FEM problems these techniques do not provide enough memory efficiency and are also not scalable with a large number of compute units. Problem specific solvers, as Fast Fourier Transformation (FFT) and cyclic reduction methods, are another direct methods [35, 45]. These methods are proven to be efficient solvers with respect to the complexity. However, they can be applied only to tridiagonal matrices in case of cyclic reduction and to Toeplitz matrices in case of FFT solver.

Due to the fact that we want to solve a very large linear system arising from an arbitrary FEM problem, we focus only on iterative solvers such as Krylov subspace and multi-grid methods.

2.4 Iterative Solvers

In this section, we present three major type of solvers – splitting methods, Krylov subspace projection methods and multi-grid methods. We focus on their convergence characteristics and implementation aspects.

2.4.1 Splitting Methods

These methods are based on a simple matrix decomposition of the original matrix A into $A = M + N$. Thus, we can define a linear fixed-point iteration scheme of the form

$$x_{k+1} = M^{-1}(b - Nx_k). \tag{2.14}$$

We need to ensure that the iteration matrix $M^{-1}N$ has spectral radius (i.e. maximum of all eigenvalues in absolute value) less than 1. In the case of regular splitting methods, where M is non-singular (i.e $\det(M) \neq 0$) and M and N are non-negative (i.e. $x^T Mx \geq 0$ and $x^T Nx \geq 0$ for all vectors x) it can be shown if A is non-singular and A^{-1} is non-negative then the spectral radius of $M^{-1}N$ is less than 1. Standard splitting schemes as Jacobi, Gauss-Seidel, Successive Over-Relaxation (SOR) and their symmetric equivalents are defined as follows

$$M_{\text{Jacobi}} := D \tag{2.15}$$

$$M_{\text{Gauss-Seidel}} := D + L \quad \text{or} \quad M_{\text{Gauss-Seidel}} := D + R \tag{2.16}$$

$$M_{\text{Symmetric Gauss-Seidel}} := (D + L)D^{-1}(D + R) \tag{2.17}$$

$$M_{\text{SOR}} := \frac{1}{\omega}(D + \omega L) \quad \text{or} \quad M_{\text{SOR}} := \frac{1}{\omega}(D + \omega R) \tag{2.18}$$

$$M_{\text{Symmetric SOR}} := \frac{1}{\omega(2 - \omega)}(D + \omega L)D^{-1}(D + \omega R) \tag{2.19}$$

where $A = L + D + R$, L is a strict lower-triangular matrix, R is a strict upper-triangular matrix, D is a matrix containing the diagonal elements of the matrix A and the relaxation parameter ω is in the interval $(0, 2)$, see [42, 114].

Typically, these methods have very bad convergence rates and are usually not used in practice [42, 114]. In some cases, the SOR method with special calculation of the relaxation parameter ω could deliver a better convergence rate which is closer to the rate of the projection methods [133]. On the other hand, these methods do not require extra data storage and they are simple to implement. As we will show later in this chapter, these splitting schemes can be successfully employed as preconditioners for Krylov subspace solvers and as smoothers for multi-grid solvers.

2.4.2 Krylov Subspace Methods

A projective iterative method is a scheme where we start with an initial solution x_0 and we look for for a new one $\hat{x} = x_0 + K_m$ such that $b - A\hat{x} \perp L_m$, where L_m and K_m are m-dimensional subspaces of $\mathbb{R}^n$. We can define the Krylov subspace methods as a subclass of the projection methods with

$$K_m := K_m(A, r_0) := \text{span}\{\mathrm{r_0}, \mathrm{Ar_0}, \mathrm{A^2 r_0}, ..., \mathrm{A^{m-1} r_0}\},$$

where $r_0 := b - Ax_0$.

In the general case of symmetric and positive definite matrices we can take $K_m = L_m$. This leads to the the best known algorithm – the Conjugate Gradient (CG) method. It can be derived in two ways from the Lanczos's algorithm, detailed information can be found in [114]. Due to the symmetry of the matrix A, a key characteristic of this algorithm is the three-term residual recurrence formula which comes from the Lanczos vectors. This allows us to construct the Krylov subspace by only keeping three vectors – p, q and r, see Algorithm 1.

The condition number of a squared matrix A can be defined, for $\lambda_{\min} \neq 0$, by

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}},$$

Algorithm 1 (Non-)Precondition Conjugate Gradient

$r_0 = b - Ax_0$ // compute initial residual
for $i = 1$ to MAX_{iter} and $||r_i||_2 \geq \epsilon||r_0||_2$ **do**
  Solve $z_i = M^{-1}r_i$ for PCG or $z_i = r_i$ for CG
  $\rho_{old} = \rho$, $\rho = r_i^T z_i$
  if $i = 1$ **then**
    $\beta = 0$
  else
    $\beta = \rho/\rho_{old}$
  end if
  $p_i = \beta p_i + z_i$ // find a new search direction
  $q_i = Ap_i$ // projection
  $\alpha = \rho/p_i^T q_i$
  $x_i = x_i + \alpha p_i$ // update the solution vector
  $r_i = r_i - \alpha q_i$ // update the residual vector ($r = b - Ax$)
end for

where $\lambda_{\max}$ and $\lambda_{\min}$ are respectively the maximal and minimal eigenvalues of the matrix A. It can be shown that the convergence rate of the CG algorithm can be expressed by

$$||x^* - x^k||_A \leq 2 \left[\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right]^k ||x^* - x^0||_A,$$

where $||x||_A := \sqrt{x^T Ax}$ is the energy norm, x^* is the true solution, x^0 is the initial value, and x^k is the k-th approximation of the solution.

In many cases, this is only a rough upper bound and in practice the method converges much faster. Lower bounds and super-linear convergence properties can be found in [18].

For the non-symmetric case one can use the Generalized Minimal Residual Method (GMRES). The main difference to the CG is that we need to build the whole Krylov space. This means that we need to build a dense matrix of size $m \times m + 1$, where m is the dimension of the Krylov space. Although that typically m is smaller than the size N of the linear system, in practice it is not feasible to store the whole basis. A remedy to that is to restart the algorithm after certain space size is reached. Details about the derivation and convergence rate of the restarted GMRES can be found in [114]. Another solver for non-symmetric matrices is the Bi-Conjugate Gradient Stabilized method (BiCGStab), see [130]. This method can be seen as a mixture between CG and GMRES methods – it requires less computation and storage than GMRES method but in some cases could lead to divergence [119].

To improve the convergence rate of the above mentioned schemes, we can decrease the condition number of the matrix and influence its spectrum by multiplying it with the inverse of an auxiliary matrix M, see [35, 42, 114]. This technique is called preconditioning, see Section 2.5. We can avoid inversion and matrix multiplication by solving a linear system based only on the matrix M. Employing a fast solver for the preconditioned system we can often dramatically decrease the execution time of the iterative solver.

2.4.3 Multi-grid Methods

Multi-grid methods are one of the most efficient solvers for elliptic problems. In this subsection, we present the concept of geometric multi-grid methods. They are based on the idea of solving the associate linear problem by tackling it on several hierarchical grid levels. First, we need to build a restriction operator which reduces a vector defined on a higher resolution grid to a vector defined on a lower resolution grid. In a similar way, we can define a prolongation operator

which interpolates a vector defined on a lower resolution grid onto a vector defined on a higher resolution grid. Doing so, we can transfer vectors defined on a fine grid to a coarse grid and vice versa. By a defect correction procedure we can transform the problem to a residual equation and solve it on a coarser grid. The idea of the multi-grid method is to solve the problem by damping the high frequency error components (smoothing phase) and solving recursively the residual problem by coarse grid approximations. We can damp the high frequency components by applying iterative schemes such as splitting methods. However, after smoothing on each level, the problem still contains low frequency error components. The multi-grid methods handle that by traversing the problem over the hierarchy of grids – going to a coarse resolution grid, the low frequency components of the error on the finer grid become high frequency error components on the coarser grid. Applying this recursively, we can reduce the problem size and then solve the system directly at the coarsest level. This leads to an asymptotic optimal linear complexity for the multi-grid methods. Theoretical and numerical results as well as practical aspects can be found in [117, 124].

Algorithm 2 Multi-grid Cycle: $u_k = \mathrm{MGC}(A_k, u_k, f_k)$

Pre-smoothing step
$u_k = \mathrm{SMOOTH}(A_k, f_k, \nu_1)$
Coarse grid restriction step
$d_k = f_k - A_k u_k$
$d_{k-1} = I_k^{k-1} d_k$
Solution step
Solve $A_{k-1} v_{k-1} = d_{k-1}$ or call $v_{k-1} = \mathrm{MGC}(A_{k-1}, v_{k-1}, d_{k-1})$
Fine grid prolongation step
$v_k = I_{k-1}^k v_{k-1}$
$u_k = u_k + v_k$
Post-smoothing step
$u_k = \mathrm{SMOOTH}(A_k, f_k, \nu_2)$

Algorithm 2 presents a multi-grid cycle. The smoothing step can be realized by applying several steps of the defect correction scheme (2.20). The number of iterations performed in the pre-processing phase is denoted by ν_1 and in the post-processing phase by ν_2. The transfer operators which restrict the defect vector from a fine to a coarser grid are I_k^{k-1}, and the ones which interpolate the correction vector from a coarse to a finer grid are I_{k-1}^k. The discrete differential operator on the k-th level is denoted by A_k. The multi-grid cycle can be executed several times recurrently until the coarsest grid system is obtained and finally, this small system can be solved directly or by a proper iterative scheme with higher precision.

The grid cycles can be traversed in different ways. Starting from a finest grid and going to the coarsest level and after that backwards the V-cycle can be defined. Better convergence properties can be obtained by multi-grid cycles with different depths [124]. The V-cycle and W-cycle are presented in Figure 2.4, where the coarsest grid is denoted by 5 and the finest by 1. In the multi-grid algorithm there are many parameters describing how many pre- and post-smoothing steps need to be performed, what kind of smoothing steps are needed, how coarse the coarsest grid should be in order to have a good representation of the prescribed geometry and a good approximation to the solution, what kind of prolongation and restriction schemes have to be used. For many applications these parameters are determined only by a heuristic approach. In these cases, the multi-grid method can be successfully used not as a solver but as a preconditioner [124].

An alternative to the geometric multi-grid method is the algebraic version of it. It can be used as a black-box solver and does not require any specification of the grid hierarchy. Two of the main drawbacks of the classical algebraic multi-grid methods are the inherited sequential

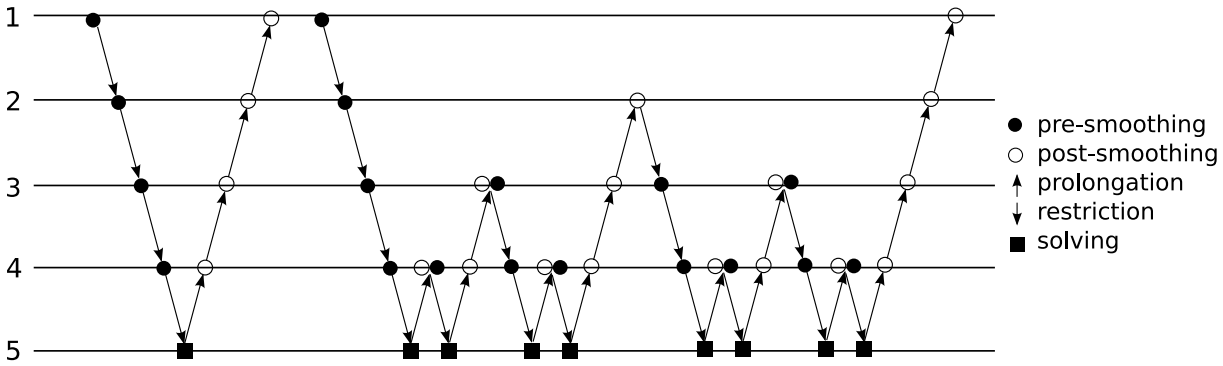

Figure 2.4: Structure of V-cycle and W-cycle

setup phase and the heuristic algorithm for building the transfer operators [124].

2.5 Preconditioners, Iterative Schemes and Smoothers

Preconditioners are used in the context of iterative solvers for decreasing the number of necessary iterations for reaching a prescribed error tolerance. This technique can be successfully used to affect the condition number of the system matrix and its spectrum.

Let us consider a linear system of the type $Ax = b$. We can modify the spectrum of the matrix A by applying a linear transformation, e.g. by multiplying the system with an auxiliary matrix M^{-1} (or a pair of matrices M_L^{-1} and M_R^{-1}). The problem reads

$$M^{-1}Ax = M^{-1}b \text{ (left preconditioner)},$$
$$AM^{-1}\hat{x} = b, \text{where } \hat{x} = Mx \text{ (right preconditioner)},$$
$$M_L^{-1}AM_R^{-1}\hat{x} = M_L^{-1}b, \text{where } \hat{x} = M_R x \text{ (split preconditioner)}.$$

In most of the iterative solvers, we can avoid sparse matrix inversions and matrix-matrix multiplications by solving an additional linear system for the matrix M. In the rest of this section we present different types of preconditioning matrices M and the ways to solve $Mz = r$. Unfortunately, there is no fundamental theory for convergence or efficiency – there is only some insight with respect to special matrices related to some well-studied equations on simple domains [11, 42].

We can classify the solving process of the preconditioning phase as explicit or implicit. The implicit type requires a solution phase of the preconditioning equation $Mz = r$. Typically, this is done by representing the preconditioning matrix in a triangular form by performing a Gaussian elimination. Splitting-type preconditioners based on additive splittings of the system matrix fall into this class. Classical schemes are Jacobi, Gauss-Seidel, their block versions and relaxed variants (e.g. SOR). Multiplicative factorizations as incomplete LU factorization maintain certain sparsity patterns or allow fill-ins into designated positions increasing the number of non-zero elements. Explicit preconditioners are the ones that directly build M^{-1}; these are the so called approximate inverse techniques. Detailed descriptions on preconditioners can be found in [35, 42, 114].

Now, we can show a connection between the preconditioning matrix M and the linear fixed-point iterative scheme. The general form of an iterative solver can be written as $x_{k+1} = Gx_k + f$. Then, for solving the linear system $Ax = b$, we can split the matrix $A = M + N$ and define the iteration matrix as $G = M^{-1}N = M^{-1}(M - A) = I - M^{-1}A$ with right-hand side $f = M^{-1}b$. Thus, we obtain the preconditioned defect correction scheme

$$x_{k+1} = x_k - \omega M^{-1}(Ax_k - b), \tag{2.20}$$

where ω is the relaxation parameter of the iteration.

In the context of multi-grid methods, the defect correction scheme combined with a preconditioner M is used as a smoothing step. A proper choice of the matrix M decreases the number of cycles in the multi-grid solver.

2.5.1 Additive Preconditioners

Preconditioning matrices based on the original structure of the matrix A are derived using Jacobi, Gauss-Seidel, Symmetric Gauss-Seidel, SOR and symmetric SOR schemes. These methods are based on the splitting technique $A = L + D + R$, where the matrix M is constructed as a combination of L, R and D, see (2.15)-(2.19). These methods are very widely spread because they do not require extra storage of the matrix and they do not need a special building procedure for the preconditioning matrix.

These standard schemes have good smoothing properties and are used in all kinds of geometric and algebraic multi-grid methods. Details can be found in [117, 124].

2.5.2 Multiplicative Preconditioners

An important class of preconditioners is based on an Incomplete LU (ILU) factorizations. Here, we are looking for a lower-triangular matrix L and an upper-triangular matrix U such that $A - M$ with $M := LU$ satisfies some criteria such as preserving certain sparsity patterns. To obtain the L and the U matrix we can perform a Gaussian elimination. In the general case, after this process, the L and the U part are not sparse, although the input matrix A is a sparse matrix. There are two algorithms for producing sparse structure of the resulting lower and upper matrices – preserving the original sparsity pattern of the matrix A or allowing fill-in entries. In Algorithm 3 we present the ILU(0) factorization without fill-ins.

Algorithm 3 Incomplete LU-factorization without fill-in elements - ILU(0)

```
for i = 2 to N do
  for k = 1 to i − 1 and (i, k) ∈ NNZ(A) do
    a_{i,k} = a_{i,k} / a_{k,k}
    for j = k + 1 to N and (i, j) ∈ NNZ(A) do
      a_{i,j} = a_{i,j} − a_{i,k} a_{k,j}
    end for
  end for
end for
```

The quality of the ILU factorization depends on the sparsity pattern of the produced L and U matrices. Therefore, to increase the accuracy of the factorization we can allow new fill-in elements in the produced matrices. A common technique to control the fill-ins is to introduce levels. Each new element of the factorization process is associated with a certain level. The ILU(p) factorization with fill-ins is presented in Algorithm 4. Details on this technique can be found in [35, 114].

One of the weak points of this algorithm is to predict the new non-zero pattern of the produced matrices for $p > 0$. Without an estimation of the distribution of the new elements, the computational costs for allocating and updating the matrix can be very high. Usually, this building process is split into two parts – symbolic step (determines the sparsity pattern) and numerical step (produces the factorization). A weighted drop-off technique can be applied to decrease the fill-in entries and to reduce the storage requirements while keeping the quality of the preconditioning matrix. Such algorithm is called ILU with threshold (ILUT factorization), see [114].

Algorithm 4 Incomplete LU-factorization with fill-in elements - ILU(p)

Set $\text{lev}(a_{i,j}) = 0$ for all non-zero elements $a_{i,j} \in \mathcal{NNZ}(A)$, $\text{lev}(a_{i,j}) = \infty$ otherwise
for $i = 2$ to N **do**
  for $k = 1$ to $i-1$ and $\text{lev}(a_{i,k}) \leq p$ **do**
    $a_{i,k} = a_{i,k}/a_{k,k}$
    for $j = k+1$ to N **do**
      $a_{i,j} = a_{i,j} - a_{i,k}a_{k,j}$
      $\text{lev}(a_{i,j}) = \min(\text{lev}(a_{i,j}), \text{lev}(a_{i,k}) + \text{lev}(a_{k,j}) + 1)$
    end for
  end for
  for $j = 2$ to N **do**
    if $\text{lev}(a_{i,j}) > p$ **then**
      delete $a_{i,j}$
    end if
  end for
end for

In general, there is no guarantee that the ILU factorization can be computed. Breakdown of the algorithm is possible if there are zero or close to zero diagonal elements of the matrix A. In order to ensure a stable factorization we can sort the zero or the closet to zero elements. This technique is called pivoting [114].

2.5.3 Approximate Inverse Preconditioners

Another preconditioners are obtained by direct approximations of the inverse matrix M^{-1}. Instead of solving the preconditioning equation $Mr = z$ we can apply only a matrix-vector multiplication with the approximate inverse matrix. Although the matrix M is sparse this cannot be guaranteed for its inverse. In this subsection, we consider a few methods for constructing a sparse approximate inverse matrix.

There are many algorithms and schemes for building the approximate M^{-1} directly. The simplest way to do this is to use the Neumann series

$$\sum_{k=0}^{\infty} J^k = (I - J)^{-1}, \tag{2.21}$$

where the spectrum radius of the matrix J is strictly less than one. We can approximate A^{-1} by shifting the spectrum of the matrix A by a parameter ω (greater or equal to the spectral radius of A) and taking $J = I - A/\omega$, see [15, 114].

Another polynomial scheme is the Chebyshev polynomial approximation. We can expand the approximate inverse matrix A^{-1} in terms of Chebyshev-matrix-values polynomials

$$A^{-1} = \frac{c_0}{2} I + \sum_{i=0}^{\infty} c_i C_i(Z), \tag{2.22}$$

where $Z = \frac{2}{\beta-\alpha}\left[A - \frac{\alpha-\beta}{2} I\right]$ is a shift matrix operator and the Chebyshev polynomials are defined by the recurrent formula

$$C_j(Z) = 2ZC_{j-1}(Z) - C_{j-2}(Z) \tag{2.23}$$

with starting values $C_0(Z) = I$ and $C_1(Z) = Z$. The coefficients α and β in the shift matrix approximate respectively the largest and the smallest eigenvalue of the matrix A in order to shift the spectrum in to the interval $[0, 1]$. The polynomial preconditioners control the sparsity

pattern of the matrix automatically. The main drawback of these schemes is that they require information about the spectrum of the matrix A.

Another technique for building the approximate inverse is to minimize the Frobenius norm

$$||I - GA||_F^2 := \sum_{i=0}^{N} ||e_i - g_i A||_2^2, \tag{2.24}$$

where e_i and g_i are respectively the columns of the identity matrix and of the approximate matrix $G := M^{-1}$. However, this does not imply any restriction on the pattern of G. Using the minimal residual iteration one can build an approximate inverse matrix. Typically, this process is cheap but the output is a dense matrix. To obtain a certain sparsity we need additional drop-off techniques [114]. In [53] the authors propose an algorithm called Sparse Approximate Inverse (SPAI) which provides an adaptive sparsity pattern. But it requires QR-decompositions and sorting procedures which are typically highly expensive.

We can obtain a set of least-squares minimization problems by prescribing a non-zero pattern $\mathcal{NNZ}_{AI}$ to the approximate inverse matrix. Thus the normal equation of (2.24) reads

$$(GAA^T)_{i,j} = A^T_{i,j} \quad \text{for } (i,j) \in \mathcal{NNZ}_{AI}. \tag{2.25}$$

This leads to a set of small and independent linear systems for each column j of the matrix G.

There are several methods for constructing a preconditioner using (2.24), see [36]. Few problems arise when constructing the preconditioners by solving directly (2.24) – there is no guarantee that the matrix is not singular and in the symmetric case the minimization problem does not necessarily build a symmetric preconditioner which is critical (e.g. for CG).

A very efficient algorithm with respect to the complexity of the building phase is the Factorized Sparse Inverse (FSAI) method. For symmetric and positive definite matrices, the scheme preserves the symmetry of the preconditioner. This method is presented in [79]. It builds an approximation not to the original matrix but to the Cholesky factorization $A = L_A L_A^T$. However, the idea can be generalized for non-symmetric matrices as well. For the symmetric case, we look for an approximation $G_L \approx L_A^{-1}$ and doing so the approximate inverse preconditioned equation GA with $G := G_L^T G_L$ reads

$$G_L^T G_L A \approx I \quad \text{or} \quad G_L A G_L^T \approx I. \tag{2.26}$$

Due to the fact that the approximation is a lower-triangular matrix we have to restrict our sparsity pattern $\mathcal{NNZ}_{AI}$ to

$$\mathcal{NNZ}^L_{AI} = \{(i,j) \in \mathcal{NNZ}_{AI} \mid (j \leq i)\}. \tag{2.27}$$

In this way, from (2.25) we obtain

$$(G_L A)_{i,j} = (L_A^T)_{i,j} \quad \text{for } (i,j) \in \mathcal{NNZ}^L_{AI}.$$

But on the other side, L_A^T is an upper-triangular matrix and by prescribing only a lower-triangular pattern we obtain only diagonal elements. This leads to the following system

$$\begin{aligned} (G_L A)_{i,j} &= 0 \quad \text{for } i \neq j, \\ (G_L A)_{i,j} &= (L_A)_{i,j} \quad \text{for } i = j. \end{aligned}$$

The most interesting and remarkable fact about this method is that we do not need to know the diagonal entries of the Cholesky decomposition. As it is described in the original work [79], we can solve the problem by taking an auxiliary matrix, applying a Jacobi scaling $G_L := D\hat{G}_L$ and

solving the new system

$$(\hat{G_L}A)_{i,j} = \delta_{i,j},$$

where $\delta_{i,j}$ is the Kronecker delta symbol. After that we can rescale with $D = \mathrm{diag}(\hat{\mathrm{G}_\mathrm{L}})^{-1/2}$. At the end, the building procedure requires only solutions of small, symmetric and positive definite problems, which can be processed in parallel. In Chapter 5, we consider a technique for building a proper sparsity pattern $\mathcal{NNZ}_{AI}$.

Another technique for building an approximate inverse is based on bi-conjugation – a generalization of the Gram-Schmidt process. The Approximate Inverse preconditioner (AINV) has a very robust building procedure. However, it is purely sequential and it requires drop-off techniques to maintain the sparsity structure, see [20, 21].

2.5.4 Other Preconditioners

2.5.4.1 Block Jacobi and Additive Schwarz Preconditioners

We can decompose the linear system $Ax = b$ into small sub-problems based on

$$A_i = R_i^T A R_i \quad \text{for } i = 0...n,$$

where R_i are restriction operators. Although the original matrix A is invertible, we cannot conclude this for the restricted matrices A_i. With respect to the different definitions of the restriction operators we can define the following schemes

- Block Jacobi – the restriction operators are built to produce only block-diagonal matrices without overlapping. Thus, all the couplings among the off-diagonal blocks are deleted, see Figure 2.5 (left).
- Additive Schwarz – the restriction operators produce sub-matrices which overlap. This leads to contributions from two (or more) preconditioners on the overlapped area, see Figure 2.5 (middle). Typically, these contribution sections are scaled by $1/k$, where k is the number of sub-domains in which the variable is represented.
- Restricted Additive Schwarz (RAS) – this is a mixture of the pure block-Jacobi and the additive Schwarz scheme. Again, the matrix A is decomposed into squared sub-matrices. The sub-matrices are large as in the additive Schwartz approach – they include overlapped areas from other blocks. After we solve the preconditioning sub-matrix problems, we provide solutions only to the non-overlapped area, see Figure 2.5 (right).

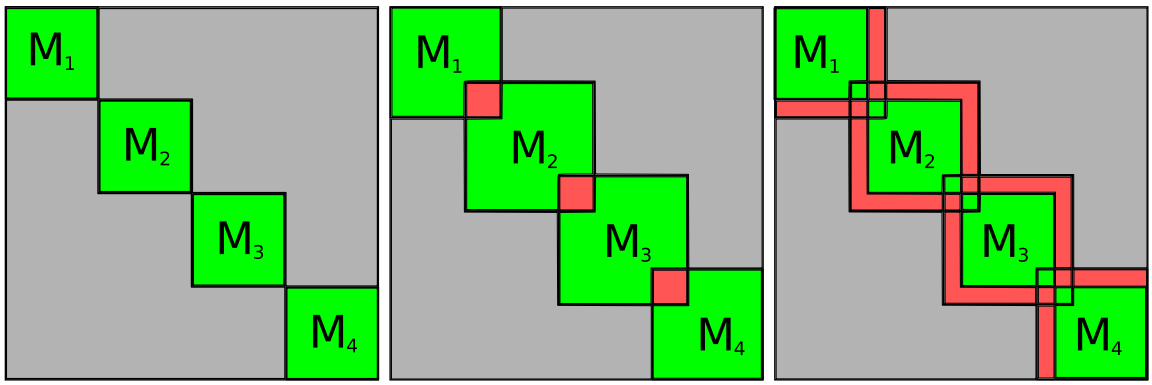

Figure 2.5: Example of a 4 block-decomposed matrix – block Jacobi preconditioner (left), additive Schwarz with overlapping preconditioner (middle), restricted additive Schwarz preconditioner (right)

Originally, the additive Schwarz methods have been developed for solving iteratively PDEs on several sub-domains. Further information could be obtained from [109, 120, 123] as well as from [114]. The RAS scheme is presented in [33]. Note that the restriction operators can be built upon information from the matrices or from the underlying grid if the linear system is based on FEM.

2.5.4.2 Support-tree/Vaidya Preconditioners

Vaidya proposed several new families of preconditioners. The first one is based on maximum spanning trees (i.e. trees that include all vertices and a subset of the edges) of the underlying matrix graph. The second approach is based on extra edges and shows a speed up in convergence. The third approach is based on a maximum weighted basis that works by dropping the non-zero elements from the coefficient matrix and factorizing the remaining matrix. The first two preconditioner families apply only to M-matrices (diagonally dominant matrices with non-positive off-diagonals elements), while the third preconditioner works only for diagonally-dominant symmetric matrices. Unfortunately, most of the original work has not been published but has been used in commercial software. An analysis and implementation of these preconditioners can be found in [34]. One of the very strong points of this type of preconditioners is the full control of the condition number of the new system. Further details can be found in the series of new papers [23, 25, 26, 27, 28].

2.5.4.3 Schur Complement Preconditioners

The main idea of the Schur complement is to divide the initial linear system into small subproblems by performing a block-Gaussian elimination. As already mentioned, in general the sub-matrices of the original matrix A are not invertible. To ensure invertibility, this method is used with domain decomposition schemes [109, 120, 123]. Dividing the original domain Ω of problem (2.1) into non-overlapping sub-domains $\Omega = \cup\Omega_i$ results into a block-linear system. The block-matrices inside the system represent the sub-domains and the couplings between them (the skeleton-matrix). In this case, we can apply a block-Gaussian elimination to reduce the problem size or to directly solve the problem. This technique can be used as a solver as well as a preconditioner. An example for application of the Schur complement as a parallel preconditioner for a flow problem can be found in [24].

2.5.4.4 Hierarchical Matrix Preconditioners

Hierarchical matrices are sub-block matrices based on sparse representations of a fully populated system [17, 32, 55]. This idea is similar to other multi-level structures such as sparse grids [131] and hierarchical basis [134]. This is in contrast to the multi-grid method where the data is built on a hierarchy of admissible partitions of the matrix indices and on a cluster-tree decomposition [17]. Preconditioners for hierarchical matrices can be built by approximate inverse and LU decomposition. The hierarchical inversion is mainly based on Schur complement decompositions and it comes at the cost of matrix-matrix multiplications. The solution process for the LU decomposed system is based on triangular solvers.

2.5.4.5 Tridiagonal Preconditioners

Tridiagonal systems can be efficiently solved by a modification of the Gaussian elimination process called Thomas Algorithm [97, 107]. Unfortunately, very few problems result in tridiagonal matrices. However, these matrices can be used for an approximation to the original linear system and thus for a preconditioning equation. We consider the tridiagonal preconditioner as a part of the splitting methods. These preconditioners can be used in the context of a multi-grid smoothers as well. Details can be found in [48].

2.5.4.6 Iterative Schemes as (Non-)Constant Preconditioners

Up to now, we have worked only with a fixed preconditioning matrix M. However, we can supply a non-constant operator M^{-1} for solving the preconditioned system. On each step of the iterative solver we can use different preconditioning schemes or we can give different approximations to the solution of the system $Mz = r$. As an example, we can use a Krylov subspace method for the inner (preconditioning) and for the outer solver. As a consequence, the outer solver needs to build a Krylov subspace that represents the new problem based on variable preconditioning matrices. Such methods are the flexible GMRES [113] and the flexible CG method [12].

We can use an inner-solver which provides a constant operator M^{-1}. Such methods are the simple iteration schemes, Chebyshev iteration and multi-grid methods, see [15, 35, 114, 124].

2.6 Algorithmic Complexity and Computational Intensity

In this section, we present a short overview and classification of the algorithmic complexity of the linear solvers that we have described so far. All of the solvers have some basic vector-vector and matrix-vector routines. To classify which routine depends on the bandwidth/communication or on the computation performance, we can define the computational intensity $I = f/w$ in asymptotic value for large data sets, where w represents the necessary data transfers (i.e. all load and store operations) and f is the number of floating point operations for executing the routine. Table 2.1 presents the performance characteristics of the main vector-vector and matrix-vector functions, where N is the vector size, $\mathcal{NNZ}$ and SM are respectively the non-zero elements and the data structure of a sparse matrix. $\mathcal{NNZ}$ denotes only the values of the non-zero elements and SM – the coordinates of the corresponding entries.

Function	w [Data]	f [Flop]	$I = f/w$
Vector norm	$N+1$	$2N-1$	2.0
Scalar product	$2N+1$	$2N-1$	1.0
Vector update	$3N+1$	$2N$	0.7
Laplace 3D stencil	$2.75N$	$8N$	2.9
Sparse matrix-vector mult	$\mathcal{NNZ}+2N+SM$	$2\mathcal{NNZ}$	< 2.0
Dense matrix-vector mult	N^2+2N	N^2-N	< 1.0

Table 2.1: Computational intensity and performance bounds for the basic linear routines

A lower bound for the total run-time T_R of an algorithm is given by $T_R \geq T_C + T_T$, where T_C is the computing time and T_T is the time for data transfers. In case of asynchronous transfers, overlapping of communication and computation, the lower bound can be taken as $\max\{T_C, T_T\}$. On a given platform an algorithm is compute-bound for $T_C > T_T$ and bandwidth-bound for $T_T > T_C$. A simple performance model can be derived by information of the algorithm and the hardware characteristics. Then we get lower bounds $T_T \geq Sw/B$ where $S = 4$ or $S = 8$ bytes for single precision or double precision data, B (in Byte/time) is the maximal bandwidth between the memory and the cores. Furthermore, for fully pipelined instructions we find $T_C \geq f/P$ where P (in Flop/time) is the accumulated theoretical peak performance of the functional units. An upper bound for the effective performance P_{eff} of the corresponding implementation can be given by

$$P_{\text{eff}} = \frac{f}{T_R} \leq \frac{f}{\max\{T_C, T_T\}} \leq \min\left\{P, \frac{fB}{Sw}\right\}.$$

For unlimited bandwidth (B very large) or compute-dominated algorithms (f very large), the upper bound is basically the peak performance P. For unlimited computing capability (P very large) the effective performance is bounded by $P_{\text{eff}} \leq fB/(Sw)$.

Taking the characteristics of the current CPU and GPU platforms (peak performance and bandwidth) and applying this formula, we can easily conclude that all vector-vector routines and matrix-vector multiplications are bandwidth bounded. No matter in what order we combine these routines, the final algorithm is bandwidth bounded. With such a model the CG algorithm is studied in [60].

In Figure 2.6 we present a classification of the most basic routines and algorithms. We define the complexity as the number of necessary floating point operations that need to be performed. We start with the basic linear algebra routines (BLAS). All of the vector-vector (BLAS1), sten-

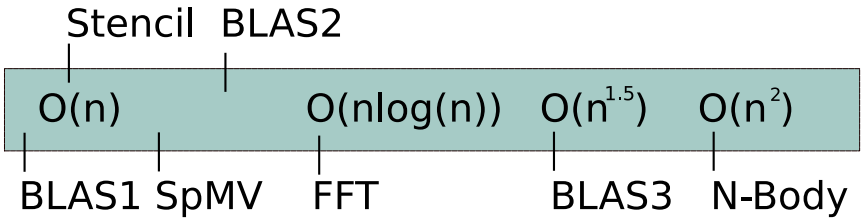

Figure 2.6: Classification of the computational complexity

cil routines, sparse and dense matrix-vector multiplications (BLAS2) have linear complexity $O(n)/O(n)$. We can construct very wide stencils with a large number of coefficients. We can build such stencils by taking higher order finite elements. In this case, for some platforms this is already a compute-bounded routine. Going further to a higher complexity of $O(nlog(n))/O(n)$ leads to a compute-bounded algorithm such as the fast Fourier transformation (FFT). One level above is the dense matrix-matrix multiplication (BLAS3) with complexity of $O(n^3)/O(n^2)$ which normalized to $O(n)$ is $O(n^{3/2})/O(n)$. Going further, the N-Body algorithms (particle physics) need to perform $O(n^2)$ operations on $O(n)$ data sets.

Chapter 3

Parallel Linear Algebra, Solvers and Preconditioners

In this chapter, we present parallel techniques for performing linear algebra routines, solvers and preconditioners. We start with the parallelism of the Basic Linear Algebra Subroutines (BLAS) including some comments and remarks on the accuracy and the consistency of results. We continue by presenting our concept for full-parallel and hybrid-parallel schemes. Special focus goes to describing the parallel execution of preconditioners based on additive and multiplicative decompositions. By formulating the forward and backward substitutions in a block-matrix form we can execute them in parallel by only performing matrix-vector multiplications without any specification of the underlying hardware or programming models. We treat the additive splitting schemes with a multi-coloring technique to provide the necessary level of parallelism. For the ILU factorization with fill-ins we propose a novel method for controlling the fill-in entries which we call the *power(q)-pattern method*. This algorithm produces a new matrix structure with diagonal blocks containing only diagonal entries. With this approach we obtain a higher degree of parallelism in comparison to the level-scheduling/topological sort algorithm. At the end of the chapter, we present remarks on other parallel preconditioning schemes.

3.1 Parallel Basic Linear Algebra Routines

As shown in Chapter 2, there are a few basic routines which are performed in almost every linear solver: vector norm, scalar product, matrix-vector multiplication, scaling and vector update functions. All of these routines and many more are standard part of an interface known as Basic Linear Algebra Subroutines (BLAS). There are libraries that provide BLAS routines, see [10, 1, 73, 99]. These basic components provide a wide variety of unique interfaces that cover almost all programming languages. These functions are also highly optimized in order to provide the most efficient utilization of the platform resources and so they are one of the basic references for performance analysis. In this section we present how to execute these routines on parallel architectures with shared memory.

3.1.1 Vector-vector Routines

All vector-vector functions are classified as BLAS1 routines. Here we show how to compute vector norm, scalar product and vector updates in parallel.

Vector updates are defined as $x = \alpha x + y$, $x = x + \alpha y$, $x = \alpha x$, $x = \alpha y$, where $x, y \in \mathbb{R}^N$ and $\alpha \in \mathbb{R}$. In these routines, there is no data dependency and therefore all components can be updated in parallel – by partially distributing the loop or by updating the values in component-wise form.

Another routines are based on reduction operations – vector norm $||x||_2$ and scalar product $x^T y$, where $x, y \in \mathbb{R}^N$. Here the sequential reduction leads to $O(N)$ operations on $O(N)$ data. In order to execute them in parallel we can introduce a partial reduction based on 2 components of the vector, see Figure 3.1 (left). The reduction needs to perform $N-1$ operations on $N = 2^k$ elements which leads to $O(log(N))$ parallel steps.

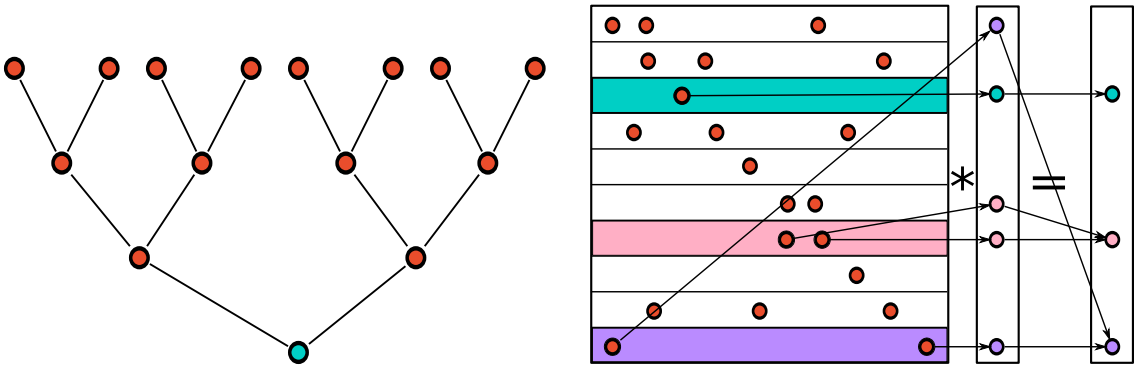

Figure 3.1: Reduction (left) and sparse matrix-vector multiplication (right) routines

We can load and compute more than 2 elements on each level. The performance profile of the parallel reductions with respect to the different blocking techniques is different on every individual hardware. Chips designed to perform loads and to execute operations on large data sets (e.g. CPUs) perform better when computing more elements per reduction cycle. On the other hand, chips which are designed to work with small data sets (e.g. GPUs) perform better when computing reduction based on a few elements.

3.1.2 Sparse Matrix-vector Routines

A typical sparse matrix produced by FEM discretization has a very low ratio of non-zero elements compared to the total size of the matrix. The most common formats for representing this sparse structure is the Compressed Sparse Row (CSR) format [15, 114]. The matrix in this format contains a row pointer array representing the offset of each row with respect to the number of elements per row. Additionally, column and value arrays represent the column indices and the element entries respectively. In the sparse matrix-vector multiplication (SpMV), each row can be computed independently of the others. Thus, a common technique is to perform all dot products of the SpMV in parallel, see Figure 3.1 (right).

Better performance for matrices with a larger number of non-zero elements per row can be obtained by using parallel reduction techniques, where two or more compute units work per row. Matrices with repeatable sparsity patterns can be represented as compressed structure of small matrices. Using this representation we can minimize the memory transfers. In order to minimize cache misses due to irregular memory access of the multiplication vector, we can introduce additional zero fill-ins. Doing so, we obtain a full-blocks structure which can improve the performance.

General remarks can be found in [15, 114], different optimization techniques for CPUs can be found in [132], and GPU implementation for different matrix formats can be found in [16, 19].

3.2 Accuracy and Consistency of the Results

One of the most important aspects in the computer floating point arithmetic is to ensure accuracy and consistency of the results. The consistency should be considered not only as reproducibility of the results on the same machine but also as reproducibility of the results from one system to another. Therefore, almost all processors used for mathematical simulations support the

IEEE Standard for Floating-Point Arithmetic (IEEE 754). This standard defines strict rules for arithmetic operations, formats, rounding and exception handling [69].

Even in the case of full IEEE 754 compliance it is hard to define reproducibility of the results since the finite arithmetic is neither an associative nor a distributive operation (i.e. $(a + b) + c$ is not $a + (b + c)$ and $a * (b + c)$ is not $a * b + a * c$).

3.2.1 IEEE standard

The IEEE 754 standard defines the way to treat floating point numbers in a finite arithmetic. In addition to the formats and the operations, this standard defines the methods used for rounding numbers during arithmetic and conversion operations. The rounding algorithms specify rounding to nearest with ties to even or ties away from zero, rounding to plus or minus infinity, and truncation rounding mode. Another characteristic of the standard is the treatment of the denormal (subnormal) numbers, the support of Not-a-Number (known as NaN) and infinity values, and handling of the overflow [49, 69, 77].

It is important to note that not all processors and accelerator boards are fully compliant with the IEEE 754 standard. There are several GPU cards that support the standard for single/double precision float-point formats only partially. Also, some processors do not support hardware division and square root, therefore these operations are handled on a software level. Therefore, if we want to make a fair comparison of numerical results produced on two machines, we have to ensure that both of the systems support the IEEE 754 standard.

3.2.2 Fused Multiply-add

In addition to the basic arithmetic operations $(+, -, *, /)$, there is also an operation which updates a value by the product of another two values $a = a + bc$, called fused multiply-add (fused-MAD). On some platforms, this operation is directly supported on the hardware level. An interesting issue is the fact that this operation has higher accuracy in comparison to the multiplication and addition functions (i.e. $d = bc$ and $a = a + d$) [69]. In order to ensure reproducibility of the results all platforms must use the same hardware routines for multiply-add operation.

3.2.3 Number of Parallel Units

The execution path of an algorithm can depend on the number of execution units (level of parallelism). Examples for routines which are independent of the number of compute units are the parallel vector updates – the order of updating the components of a vector is irrelevant to the final result. However, the parallel reduction operations in finite floating arithmetic as vector norm and scalar product depend on the partitioning of the data which relies on the number of execution units. Working with vectors that are filled with sensitive data (e.g. very large and very small values) we obtain different results for sequential or parallel reduction due to the different propagation of the rounding errors. This implies that if we compute a vector norm sequentially on a single core system and compute the same norm on a GPU device with hundreds of execution units in parallel, the results could be different even if both of the platforms have identical floating point arithmetic.

3.2.4 Hardware Errors and ECC Protection

Hardware errors could appear in a computer system due to overheating, cosmic rays, electric, magnetic or radiation fields. In practice these errors are very hard to detect and handle. However, some vulnerable parts of the computer such as the memory can be protected by using special algorithms that check the correctness of the memory transactions. Error Correction Codes (ECC) is such a mechanism – it works by introducing check sums over the transferred data for correctness

control. On the modern GPU devices we can enable and disable the ECC control. The use of ECC results in memory bandwidth and capacity decreasing [103].

3.2.5 Impact on the Basic Routines and Solvers

Clearly, any floating point error has impact on the basic routines and on the solvers as well. In Table 3.1 we summarize the influence of the different errors.

	Scalar product/Vector norm	SpMV/Stencil	Vector-updates
Hardware-errors	Yes	Yes	Yes
IEEE-754	Yes	Yes	Yes
# Parallel units	Yes	No/Yes	No
Fused-MAD	Yes	Yes	Yes
Loop-reorganization	Yes	No/Yes	No
Re-ordering techniques	Yes	Yes	No

Table 3.1: Impact of different floating point errors on the basic linear routines

Re-ordering the degrees of freedom of the discrete problem leads to different error propagation in the scalar product, the vector norm and the SpMV routines. Reduction routines are typically performed in a single or group form of component-wise loops and thus, reorganization of these loops leads to a different error propagation. The accuracy of the multiply-add operation depends on the use of fused routine. The error propagation of the routines whose execution path depends on the number of parallel units (i.e. reduction operations such as vector norm, scalar products and in some cases SpMV) is a function of their number.

Perturbed data and results in the basic linear algebra routines also have impact on the solvers. Thus, the reproducibility of the results is a combination of the hardware specifications, properties of the solver and characteristics of the discrete problem.

3.3 A Concept for Parallel Solvers and Preconditioners

3.3.1 Flexibility, Portability and Scalability

Our concept is not to provide an isolated parallel implementation for a specific solver but to establish a general framework for developing, supporting and maintaining preconditioned iterative solvers and multi-grid methods. Based on our experience, it is a great challenge to develop such a solver. But it is an even greater challenge to develop it in a portable and flexible way – without modifying the solver code and by only supplying add-ons to execute the solver on a new parallel platform. However, this should not come at the cost of poor efficiency and scalability.

In order to fulfill all requirements, we propose a concept to build all considered solvers and preconditioners only on the top of vector-vector, matrix-vector and matrix-matrix operations which are highly parallel.

3.3.2 Full-parallel Schemes

Most of the iterative solvers can be built only with the basic linear algebra routines – such solvers are CG, BiCGStab, Chebyshev-iteration and others. If we assume that we can solve the preconditioning equation (based on additive, multiplicative and approximate inverse schemes) in the same way, then we can include as well preconditioned CG, BiCGStab, additive (splitting) iterative schemes and multi-grid methods.

As already mentioned, most of the solvers can be built on the top of basic linear algebra routines which can be executed in parallel. The main effort in this direction is to provide full parallel solvers for the preconditioning matrix – this is addressed in Section 3.4

3.3.3 Hybrid-parallel Schemes

Due to the specific solver requirements, we can perform platform-optimized routines as an execution of sequential tasks. An example for that is the execution of the GMRES solver – based on Givens rotations, it builds the Hessenberg matrix sequentially. The solver can be performed on a hybrid architecture (e.g. GPU-CPU platform) by performing the vector-vector and matrix-vector routines on a highly parallel platform and computing the Hessenberg matrix on the sequential-optimal device. Another example is the multi-grid method, where one could perform a direct solver (typically a sequential process) on the coarsest level on the CPU device.

3.4 Parallel Preconditioners

In the following sections, we present several methods and techniques which adapt or construct the sparsity structure of the matrix M in order to solve the preconditioning system $Mz = r$ efficiently and in parallel. Our goal is to obtain highly parallel preconditioners without sequential parts that can be used as black-box algorithms on parallel hardware platforms. From a practical point, it is important to decrease not only the number of iterations but also the total time for the iterative solver. Therefore, it is a key point to perform the preconditioning step as fast and as cheap as possible while maintaining the iteration reduction property.

3.5 Block Jacobi and Additive Schwarz Preconditioners

As we have shown in Section 2.5.4.1, the block Jacobi, restricted additive Schwarz (RAS) or general additive Schwarz decouple the preconditioning matrix into sub-blocks. The obtained sub-matrices can be computed in parallel. The level of parallelism is given by the number of blocks in these schemes. The Jacobi preconditioner is one of the simplest preconditioners and can be derived by increasing the number of blocks in the block-Jacobi case to the size of the matrix.

The main disadvantage of these schemes is the lack of scalability – as the number of blocks increases (i.e. the levels of parallelism) the coupling and thus the quality and the efficiency of the preconditioner decreases, see [61].

3.6 LU-based Preconditioners/Block-wise Sweeps

We propose to decompose the matrix into smaller blocks. In our context, the level of parallelism is not determined by the number of blocks but by the number of elements per block. Our concept for LU-based parallel preconditioners is to iterate over the blocks on the diagonal in a sequential manner, where each block or block row is processed in parallel. The number and the size of the blocks in the decomposition is derived by analyzing the matrix structure. We use matrix re-ordering schemes in order to identify maximal independent sets of nodes and to eliminate dependencies by multi-coloring or level-scheduling algorithm (which are considered later in this section). Matrix decompositions are either additive (splitting-type methods) or multiplicative (ILU-type methods). In the latter case, there is a pre-processing step for producing the matrix factorization. Moreover, matrix re-orderings are performed in the pre-processing phase as well. The idea of the matrix re-ordering schemes (also called permutation) is to re-number the nodes of the matrix in order to obtain a new non-zero pattern. In some cases, during the factorization process additional fill-ins are permitted and are a necessary building block for maintaining matrix couplings. Special measures are taken to control the fill-in-pattern and to prevent fill-ins into the diagonal blocks.

To solve a triangular linear system, we need to perform a forward substitution. As an example, in Figure 3.2 we present a lower-triangular matrix of size 3-by-3.

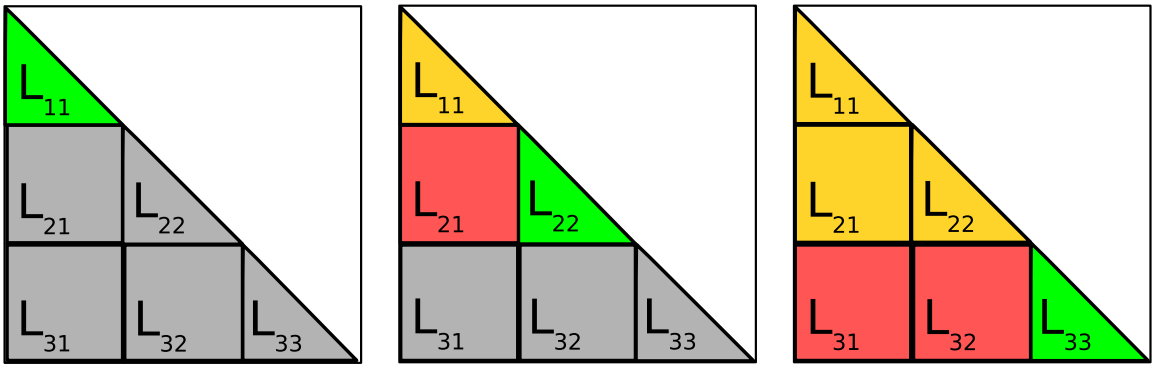

Figure 3.2: Example of a 3-by-3 block-decomposed matrix with element/block-wise elimination – with green we denote the inversion of the element/block; with red the elements/blocks which are subtracted during the substitution; with orange – the already computed elements/blocks

We want to solve $Lz = r$ for some given vector r. Thus we can perform the following three steps

$$
\begin{aligned}
z_1 &= L_{11}^{-1}(r_1), \\
z_2 &= L_{22}^{-1}(r_2 - L_{21}z_1), \\
z_3 &= L_{33}^{-1}(r_3 - L_{31}z_1 - L_{32}z_2).
\end{aligned}
$$

Note that this works not only for matrices of size 3-by-3 but also for block-matrices with 3-by-3 blocks. The only difference is in the inversion of the diagonal blocks L_{11}, L_{22}, L_{33}. In the classical forward substitution this is performed in a point-wise manner, while the block-wise version is performed by matrix-vector multiplications and inversions. In the latter case, the major difficulty is to provide an efficient inversion of the diagonal blocks.

3.6.1 Additive Preconditioners

For splitting-type preconditioners we choose a block decomposition $A = D + L + R$ where $D := \mathrm{diag}(D_1, \ldots, D_B)$ with square matrices D_i of size $b_i \times b_i$, $i = 1, \ldots, B$, L is a strict lower triangular matrix and R is a strict upper triangular matrix. The decomposition of the system matrix A into 4 blocks ($B = 4$) is shown in Figure 3.3 (left).

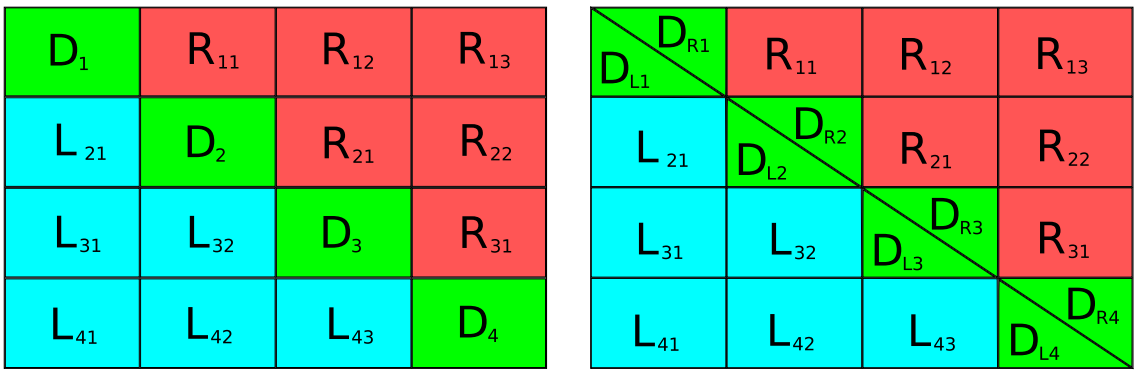

Figure 3.3: Example of a 4-by-4 block-decomposed matrix – additive splitting for Gauss-Seidel-type methods (left) and multiplicative splitting for ILU-type methods (right)

Thus, for the symmetric block-Gauss-Seidel (SGS) preconditioner we choose $M := (D + L)D^{-1}(D + R)$ and solve the preconditioning equation $Mz = r$ by the sequence $(D + L)x = r$,

$D^{-1}y = x$ and $(D+R)z = y$. This translates to

$$x_i = D_i^{-1}(r_i - \sum_{j=1}^{i-1} L_{i,j}x_j) \quad \text{for } i = 1,\ldots,B, \tag{3.1}$$

$$y_i = D_i x_i \quad \text{for } i = 1,\ldots,B, \tag{3.2}$$

$$z_i = D_i^{-1}(y_i - \sum_{j=1}^{B-i} R_{i,j}z_{i+j}) \quad \text{for } i = B,\ldots,1, \tag{3.3}$$

with block vectors r_k, x_k, y_k and z_k of length b_k, $k = 1,\ldots,B$.

3.6.2 Sweeps Granularity

The bracket expressions on the right-hand sides of (3.1) and (3.3) now consist of $i-1$ and $B-i$ matrix-vector products with vectors of length b_j and b_{i+j}, respectively. In total B^2 sparse matrix-vector products and $2B$ sparse matrix inversions are necessary to compute (3.1)-(3.3). The vectors x_k, y_k and z_k, $k = 1,\ldots,B$ are block vectors of length b_k. For a block row-wise execution the degree of parallelism in each step is b_i (assuming parallel inversion of D_i). The value of b_i is N/B for a uniform block size and is typically much larger than B.

The blocks in a block row can either be processed in a single step by means of a specific kernel or can be processed one after another by calling standard SpMV and BLAS routines. Sequential usage of data parallel BLAS1 and BLAS2 routines on each block has the same degree of parallelism, i.e. the height b_k of the block row. Although, instead of one kernel for the whole block row, B SpMV kernels are called, the work complexity stays the same. We emphasize that the usage of standard routines is a key point for a flexible implementation. This design decision keeps maximal portability with respect to new platforms.

3.6.3 Multi-coloring and Parallel Sparse Triangular Solvers

The inversion of the diagonal blocks D_i can be easily done when the blocks contain only diagonal elements. Thus, we are looking for a permutation π of the matrix A which produces a block-decomposed matrix with only diagonal elements in the diagonal sub-matrices. This can be achieved by applying the multi-coloring algorithm as a preprocessing step. Triangular solvers are then reduced to the inversion of diagonal matrices and matrix vector products, both of which can be performed in parallel on each block level with a high degree of parallelism.

The basic idea of the multi-coloring approach is to resolve dependencies between neighboring elements by introducing neighborship classes (colors) such that for non-zero matrix elements $a_{i,j} \in \mathcal{NNZ}(A)$ with $A = (a_{i,j})_{i,j=1,\ldots,N}$ both indices i and j are not members of the same class (color). A straightforward greedy algorithm for determining the colored index sets is Algorithm 5, see [114]. Here, $\text{Adj}(i) = \{j \neq i \,|\, a_{i,j} \neq 0\}$ are the adjacent non-zero nodes to node i. By re-

Algorithm 5 Multi-coloring

```
for i = 1 to N do
  Set color(i)=0
end for
for i = 1 to N do
  Set color(i)=min(k > 0 : k ≠color(j) for j ∈ Adj(i))
end for
```

numbering the nodes by colors, the diagonal blocks D_i become diagonal. Here B is the number of colors and b_k is the number of elements of color k. The inversion of the diagonal matrix then is only a component-wise scaling of the source vector. Due to the data parallelism of the associated

vector and matrix-vector routines there is no load imbalance even for varying block sizes (unless the number of elements per block is too small compared to the number of parallel units). Based on the index colors we can build π by re-ordering the nodes by groups of colors.

3.6.4 Multiplicative Preconditioners – Incomplete LU Factorization

A similar idea applies to ILU decompositions. Here, the matrix is decomposed into a product $A = LU + R$ of a lower triangular matrix L, an upper triangular matrix U and a remainder matrix R. Typically, the diagonal entries of L are taken to be ones and both matrices are stored in the same data structure (omitting the ones). As before, the matrix is further decomposed into blocks as illustrated in Figure 3.3 (right).

In the preconditioning step, i.e. when solving $Mz = r$ with $M := LU$, we have to perform two triangular sweeps – the forward step $Lx = r$ for the L-part and the backward step $Uz = x$ for the U-part. We can re-write these classical LU sweeps in block matrix-vector form by

$$x_i = D_{Li}^{-1}(r_i - \sum_{j=1}^{i-1} L_{i,j}x_j) \quad \text{for } i = 1, \ldots, B, \tag{3.4}$$

$$z_i = D_{Ri}^{-1}(x_i - \sum_{j=1}^{B-i} R_{i,j}z_{i+j}) \quad \text{for } i = B, \ldots, 1. \tag{3.5}$$

The major difficulty in computing (3.4) and (3.5) arises from solving the diagonal blocks D_{Li} and D_{Ri}, which are usually non-diagonal themselves. In the following sections, we present different algorithms and techniques for handling these issues.

3.6.5 ILU(0) with Sparsity Pattern Based on the Original Matrix

Let a matrix decomposition $A = LU + R$ be given with some sparse remainder matrix R. The occupancy pattern of L and U in the ILU(0) decomposition is chosen such that no additional elements are inserted into originally unpopulated positions. Additional elements are offloaded to the remainder matrix R. We are looking for a permutation π that re-arranges A in such a way that we obtain only diagonal elements in its diagonal blocks D_{Li} and D_{Ri} (cf. Figure 3.3 (right)). With this new representation, only easy matrix inversions in (3.4) and (3.5) have to be done. The problem of finding π can be solved by using the multi-coloring Algorithm 5 applied to A.

3.6.6 ILU(p) with Fill-in Elements

The approximation quality of the ILU factorization depends on the sparsity pattern of the resulting matrices. Therefore, in order to increase the quality of the factorization we can allow further fill-in elements in the factorization matrices. A common technique to control the fill-ins is to introduce levels. Each new element of the factorization process is associated with a certain level p, see Algorithm 4, Section 2.5.2. The produced factorized matrices associated with the p-level of fill-ins are denoted by L_p and U_p.

A difficulty that arises with the fill-ins in the algorithm is to predict the new non-zero pattern of the resulting factorization matrices for $p > 0$. There should be no fill-ins into the diagonal blocks. Without preliminary information on the distribution of the inserted elements, the costs for allocating the memory and updating the matrix during the factorization construction by means of dynamical data structures can be significant. Unfortunately, there is no general answer about the way in which the multi-coloring method affects the locality structure of a specific matrix. In most implementations, the incomplete factorization with fill-ins is constructed in two phases – algebraic step (determines the sparsity pattern) and numerical step (performs the

factorization). There are few works discussing the behavior of the matrix structure based on levels and the way to build this structure in parallel, see [65, 116]. As before, our aim is to provide an easy inversion of the diagonal blocks D_{Li} and D_{Ri} which are non-diagonal in general.

3.6.7 Level-scheduling Algorithm

We want to perform the forward step (3.4) and the backward step (3.5) for the ILU(p) factorization with fill-ins in parallel. However, due to the fill-ins, the sparsity patterns of L_p and U_p do not correspond anymore to the sparsity pattern of A. Multi-coloring on the level of the L_p and U_p matrices cannot be applied because this would destroy their upper and lower diagonal structure (no exchange of elements is allowed over the diagonal). Therefore, we need to sort the unknowns in such a way that the i-th equation depends only on the previous $i-1$ unknowns. This method is called *topological sorting*. The *level-scheduling* algorithm proposed in [114] is based on this method and has linear time complexity. The level-scheduling algorithm for a lower triangular matrix given in Algorithm 6 defines levels of depth for all rows $i = 1, \ldots, N$ corresponding to a node i in the adjacency graph and to the variable i, respectively. Then we can

Algorithm 6 Level scheduling algorithm

Let $A = (a_{i,j})$ be a lower triangular matrix
for $i = 1$ to N **do**
  $\text{depth}(i) = 1 + \max_j\{\text{depth}(j) \text{ for all } j \text{ with } a_{i,j} \neq 0\}$
end for

create a permutation matrix π that groups all the nodes with the same depth. The degree of parallelism is given by the number of elements per block, i.e. the number of nodes with the same depth, where the matrix is processed block after block.

Simple tests on a suite of matrices show that the number of levels produced by the level-scheduling algorithm is quite high, see Section 5.3.3. Slightly better results can be obtained if the ILU(p) factorization is preformed after a multi-coloring step and level-scheduling is applied afterward. This additional step decreases the number of levels in comparison to the version without multi-coloring permutation. However, in many cases this improvement is not significant.

In the general case, where $\mathcal{NNZ}(L_p) \neq \mathcal{NNZ}(U_p^T)$, the decomposition requires two permutations – π_1 for the L_p part and π_2 for the U_p part. This is a necessary step, otherwise the solutions of the L_p and U_p sweeps are incompatible. Thus, we need to permute the matrix A (or make indirect indexing) during the preconditioning phase. This is in contrast to the multi-colored ILU(0) where the permutation is appliesd as a pre-processing step.

To the best of our knowledge, there are few works on that, see [98, 114, 115]. Regardless of the implementation, the number of levels of the resulting matrix is relatively high which decreases the level of parallelism significantly. In our approach, the number of SpMV in the triangular ILU solver scheme is $B^2 - B$, where B is the number of blocks (i.e. levels) which can be processed in parallel. Therefore, the number of levels should be kept as low as possible (to prevent function call overheads for small sub-matrices) and the number of elements per level should not be too small.

3.6.8 *Power(q)-pattern Method*

For the parallel solution of the ILU(p) sweeps with fill-ins we propose the *power(q)-pattern method*. The main idea is to produce a block matrix structure with only diagonal elements in the diagonal blocks after the factorization. In this subsection we derive an upper bound for the non-zero pattern of the modified matrix factorization.

The produced non-zero pattern of the ILU(p) factorization matrix with level-based fill-ins looks very similar to the sparse matrix-matrix multiplication pattern. This leads to the observation that the non-zero sparsity pattern after the factorization of ILU(p) grows like $|A|^{p+1}$, where

$|.|$ denotes the element-wise modulus of the matrix, i.e. $|A| = (|a_{i,j}|)$. Inspired by this fact, we can restrict the non-zero pattern of the factorization to the pre-determined pattern of $|A|^{p+1}$, see [62]. A modification of the original algorithm is presented in Algorithm 7.

Algorithm 7 Power(q)-pattern enhanced ILU(p,q) with $q = p + 1$

```
Determine sparsity pattern NNZ(|A|^{p+1}) of matrix power |A|^{p+1}
Set lev(a_{i,j}) = 0 for all non-zero elements a_{i,j} ∈ NNZ(A), lev(a_{i,j}) = ∞ otherwise
for i = 2 to N do
  for k = 1 to i − 1 and (i, k) ∈ NNZ(|A|^{p+1}) with lev(a_{i,k}) ≤ p do
    a_{i,k} = a_{i,k}/a_{k,k}
    for j = k + 1 to N and (i, j) ∈ NNZ(|A|^{p+1}) do
      a_{i,j} = a_{i,j} − a_{i,k}a_{k,j}
      lev(a_{i,j}) = min(lev(a_{i,j}), lev(a_{i,k}) + lev(a_{k,j}) + 1)
    end for
  end for
  for j = 2 to N do
    if lev(a_{i,j}) > p then
      delete a_{i,j}
    end if
  end for
end for
Delete all entries a_{i,j} where a_{i,j} = 0 (compress the matrix due to possible erasement)
```

This variation of the original ILU(p) scheme (cf. Algorithm 4) ensures that fill-ins up to level p only appear in positions determined by the sparsity pattern of $|A|^{p+1}$. Moreover, in comparison to the original ILU(p) algorithm, the two inner loops are restricted to a few values that are known in advance. Consequently, building the ILU decomposition can be done much faster. There is no more need to run the full inner loops and to insert elements in a dynamic data structure. And not less important, by constructing the factorization in this way we have a full control over the sparsity patterns of the factor matrices L_p and U_p. More precisely, we find:

Proposition 3.6.1. *Let L_p and U_p be the output of the power(q)-pattern enhanced ILU(p,q) decomposition of the matrix A with $q = p + 1$ as detailed in Algorithm 7. Then we have $\mathcal{NNZ}(L_p) \cup \mathcal{NNZ}(U_p) \subseteq \mathcal{NNZ}(|A|^{p+1})$.*

Proof. The assertion follows by construction. New elements in L_p and U_p can only occur in positions already populated in $|A|^{p+1}$. □

In case $p = 0$ or $p = 1$ there is no difference in the factorization results between Algorithm 7 and the original ILU(p) Algorithm 4. For $p \geq 2$ the original algorithm might produce slightly larger non-zero patterns for general sparse matrices. However, for all of the cases we study, the power(q)-pattern enhanced ILU(p,q) algorithm with $q = p + 1$ produces the same matrix factors as the original ILU(p) algorithm.

By the next proposition we see that the matrix pattern can be further influenced and controlled by a multi-coloring step.

Proposition 3.6.2. *Let $A = (a_{i,j})_{i,j=1,\ldots,N}$ be a matrix with non-zero elements on its diagonal, i.e. $a_{i,i} \neq 0$ for $i = 1, \ldots, N$. Let π be the permutation matrix based on the multi-coloring algorithm applied to the matrix $|A|^q$ for an integer $q \geq 1$. Then, for every positive integer $l \leq q$, the matrix transformation $\pi|A|^l\pi^{-1}$ results in a block-decomposed matrix where the diagonal blocks have non-zero elements on their diagonals only.*

Proof. With π given by the multi-coloring permutation for input $|A|^q$, we define $\hat{A} := \pi|A|^q\pi^{-1}$. This is a block-decomposed matrix with only diagonal elements in its diagonal blocks. With

$\tilde{A} := \pi|A|\pi^{-1}$ we find $\tilde{A}^q = \hat{A}$ and $\tilde{A}^l = \pi|A|^l\pi^{-1}$ for all l. We also find $|\tilde{A}| = \tilde{A}$. If a positive matrix element $b_{n,m}$ of a non-negative matrix $B = (b_{i,j})_{i,j=1,\ldots,N}$ is given and the non-negative matrix $C = (c_{i,j})_{i,j=1,\ldots,N}$ has positive diagonal elements, i.e. $c_{k,k} > 0$ for all k, then the corresponding element $(BC)_{n,m}$ of the matrix product BC is positive due to $(BC)_{n,m} = \sum_k b_{n,k}c_{k,m} \geq b_{n,m}c_{m,m} > 0$. By this, for l, q integers, we conclude

$$\mathcal{NNZ}(\tilde{A}) = \mathcal{NNZ}(|\tilde{A}|) \subseteq \mathcal{NNZ}(|\tilde{A}|^l) = \mathcal{NNZ}(\tilde{A}^l) \subseteq \mathcal{NNZ}(|\tilde{A}|^q) \quad \text{for all } 0 < l \leq q$$

and so we find

$$\mathcal{NNZ}(\pi A\pi^{-1}) \subseteq \mathcal{NNZ}(\pi|A|^l\pi^{-1}) \subseteq \mathcal{NNZ}(\pi|A|^q\pi^{-1}) \quad \text{for all } 0 < l \leq q.$$

Hence, $\pi A\pi^{-1}$ and $\pi|A|^l\pi^{-1}$, $1 \leq l \leq q$, are block-decomposed matrices where the diagonal blocks have only diagonal elements. □

Now, we combine Proposition 3.6.1 and Proposition 3.6.2 to formulate

Proposition 3.6.3. *Let $A = (a_{i,j})_{i,j=1,\ldots,N}$ be a matrix with all non-zero elements on its diagonal, i.e. $a_{i,i} \neq 0$ for $i = 1, \ldots, N$. Let π denote the multi-coloring permutation based on the matrix $|A|^{p+1}$ and $A_\pi := \pi A\pi^{-1}$ be the resulting block-decomposed matrix with only diagonal elements in its diagonal blocks. Then the power(q)-pattern enhanced ILU(p,q) factorization with $q = p + 1$ given by Algorithm 7 applied to A_π produces two block-decomposed factorized matrices L_p and U_p which have only diagonal elements in their diagonal blocks. Fill-ins occur only outside the diagonal blocks.*

Proof. By Proposition 3.6.2 the matrix A_π is a block-decomposed matrix with only diagonal elements in its diagonal blocks. By applying Algorithm 7 to A_π we obtain matrix factors L_p and U_p with $\mathcal{NNZ}(L_p) \cup \mathcal{NNZ}(U_p) \subseteq \mathcal{NNZ}(|A_\pi|^{p+1})$ due to Proposition 3.6.1. Since $|A_\pi|^{p+1} = \pi|A|^{p+1}\pi^{-1}$ has off-diagonal elements equal to zero in its diagonal blocks, both matrices L_p and U_p have no fill-ins in its diagonal blocks. □

Application of Proposition 3.6.3 results in the following Algorithm 8, the *power(q)-pattern enhanced multi-colored ILU(p,q) method*. In the general case q is taken as $q = p + 1$. Later, we consider also the case $0 < q \leq p$.

Algorithm 8 Power(q)-pattern enhanced multi-colored ILU(p,q) method with re-arranged fill-ins for parallel triangular sweeps

Building of power(q)-pattern enhanced multi-colored ILU(p,q)
Perform multi-coloring analysis for $|A|^q$ with $q = p + 1$ and obtain
- corresponding permutation π
- the number of colors B
- local block sizes b_i

Permute $A_\pi := \pi A\pi^{-1}$
Apply modified ILU(p,$p+1$) factorization (cf. Algorithm 7) to A_π
Obtain factor matrices L_p and U_p with only diagonal elements in diagonal blocks
–no further fill-ins into diagonal blocks

Perform parallel forward/backward sweeps
Perform parallel triangular sweeps(3.4) and (3.5)
- use given number of colors B and local block sizes b_i

This algorithm produces a block-decomposed system that constructs the ILU problem for parallel execution of the forward and backward sweeps. Compared to the original multi-coloring scheme (applied to A instead of $|A|^{p+1}$) further couplings are maintained by fill-in elements

(outside of diagonal blocks) and additional colors are used. But in practical applications, the number of colors is typically much lower than the one obtained by the level-scheduling algorithm.

3.6.8.1 Building Phase Complexity

As in many algorithms for fast incomplete factorization (see [13, 65, 81, 116]), we also split the process by performing an algebraic phase and a numerical factorization. However, in the proposed power(q)-pattern enhanced multi-colored ILU decomposition scheme, the factorization corresponds only to the building of the sparsity pattern of the matrix $|A|^q$. To perform this step in parallel is simpler than to perform the algebraic phase of the ILU(p) factorization. Details for sparse matrix-matrix multiplication algorithms can be found in [13, 14].

After the new matrix structure is determined, the numerical factorization is performed. For the standard schemes and for the power(q)-pattern method, this procedure is the same. This process has linear complexity with respect to the number of non-zero entries produced in the algebraic phase.

In previous work for symmetric problems [65, 81], it has been shown that the complexity of the process which produces the new structure is linear with respect to the number of non-zero entries in the factorized system. In the power(q)-pattern method, the production of the new sparse matrix is also linear with respect to the number of unknowns – for the sparse matrix-matrix multiplication we follow [14]. Furthermore, [65, 81] show that the complexity is not optimal for non-symmetric matrices. This is not the case in the power(q)-pattern method – the complexity of the non-symmetric and the symmetric case is based only on the sparse matrix-matrix multiplication (which is linear with respect to the new non-zero elements).

3.6.8.2 Pivoting

Another difference when comparing the original ILU algorithm to the power(q)-pattern method is the permutation of the matrix before the factorization, which is based on the topology of the graph and not on the actual values.

One of the drawbacks of the ILU decomposition is the possible breakdown of the procedure when pivoting is not applied [114]. For some matrix types (e.g. diagonally dominant matrices), a proper processing of the decomposition is ensured without pivoting. Point-wise pivoting is not possible for the proposed power(q)-pattern enhanced multi-colored ILU(p, q) scheme because this destroys the block-diagonal structure of the factorization. In the power(q)-pattern method, we can apply a permutation based on the multi-coloring decomposition of the factorization – we are free to order the sub-blocks in the matrix as we want. The blocks could be selected by introducing a norm over the sub-blocks.

3.6.8.3 Cholesky Decomposition

The proposed power(q)-pattern method can also be applied for obtaining the symmetric Cholesky decomposition described in [114]. The sparsity pattern is produced in the same manner, only the factorization process has to be adapted for the symmetric case. We need to solve $Mz = r$, where $M := LL^T$ and thus we can re-write the substitutions as follows

$$x_i = D_{Li}^{-1}(r_i - \sum_{j=1}^{i-1} L_{i,j}x_j) \quad \text{for } i = 1, \ldots, B, \tag{3.6}$$

$$z_i = (D_{Li}^T)^{-1}(x_i - \sum_{j=1}^{B-i} L_{i,j}^T z_{i+j}) \quad \text{for } i = B, \ldots, 1. \tag{3.7}$$

Here, the factorized matrix has the same structure as in Figure 3.3 but we work only with the lower-triangular part of it.

3.6.8.4 Another Viewpoint of Sparsity Pattern

Algorithm 7 can be seen as control of a pattern similar to the pattern growth of the approximation to the inverse matrix based on matrix-valued Chebyshev polynomials (2.23). In the power(q)-pattern method, the produced structure after the factorization is a subset of the $|A|^{p+1}$, see Proposition 3.6.1. The approximate inverse Chebyshev polynomial grows in a similar way, namely the pattern can be expressed by $\sum_{i=1}^{p+1} A^i$ which can be deduced from (2.22).

3.6.9 Increasing Parallelism by Drop-off Techniques

We can increase the degree of parallelism by deleting selected elements on designated places of the obtained factorization matrices L_p and U_p. We address two techniques – one for the level scheduling algorithm, and another one for the power(q)-pattern enhanced multi-colored ILU(p,q) method. Of course, deletion of a large number of elements should be handled carefully because it comes at the expense of preconditioning efficiency and the convergence of the iterative solver is typically harmed.

Level-scheduling The elements to be deleted can be selected within a given radius from the main diagonal of a predefined matrix structure (e.g. given by multi-coloring permutation applied to the original input matrix). In this case we can define a threshold value so that all elements below this threshold are to be deleted. As an observation, this technique increases the level of parallelism only a little but the efficiency of the preconditioner is decreased in most of the cases.

Power(q)-pattern The number of colors in the power(q)-pattern enhanced ILU(p,q) method can be artificially decreased by choosing a smaller exponent $q < p + 1$ for determining the upper bound of the sparsity pattern. With multi-coloring based on $|A|^q$ with $q < p + 1$ and the modified ILU(p) applied, fill-ins into the diagonal blocks are possible. These fill-in elements are selected for deletion. The effect of the drop-off strategy on the number of iterations for the CG solver and the non-zero pattern of the factorized matrix for some sample matrices is presented in Section 5.3.3. Note, that the number of SpMV grows quadratically with respect to the number of colors obtained for the forward and backward steps (of course, the size of the matrices gets smaller). Therefore, the choice $q = p+1$ is no longer suitable for large p. For GPU computations, there is a considerable overhead for invoking a huge number of kernels (e.g. for matrix-vector operations) and thus the number of operations should be kept low, see Chapter 5.

3.6.10 Comparison of Power(q)-pattern and Level-scheduling Method

Building step Both of the algorithms work in a different way. The level-scheduling algorithm is applied after the factorization process as an analyzing step of the elimination dependency while the power(q)-pattern enhanced ILU(p, q) method builds the factorization structure in a particular way before performing the factorization step.

Degree of Parallelism The power(q)-pattern enhanced ILU(p, q) method is based on a multi-coloring decomposition of the structure of the matrix $|A|^q$. For all $q \geq 0$, this matrix represents a form of topological connectivity for a differential operator and a discretization scheme. This decomposition is independent of the problem size, i.e. the connectivity pattern is not a function of the grid size. However, the level-scheduling algorithm relies on the elimination process of the triangular system which is a function of the problem size. Thus, we can conclude that by increasing the size of the problem the level-scheduling method keeps decreasing the level of parallelism while the power(q)-pattern enhanced ILU(p, q) method keeps it constant. A numerical example underlining this observation is considered in Section 5.3.6 and 5.4.6.

Permutation As pointed out, in general, after the factorization the non-zero patterns of the lower and the upper triangular matrices can be different. This requires extra work for the level-scheduling algorithm. The level-scheduling solver has either to perform irregular access over a special index set which describes the dependency (this produces an overhead) or to permute the vectors twice (which is expensive). On the other hand, the power(q)-pattern enhanced ILU(p, q) method requires only one single permutation which can be performed as a pre-processing step.

Pivoting In the case of level-scheduling scheme, to avoid breakdown of the factorization due to zero or close to zero elements, the user can apply elements pivoting before the factorization process while in the power(q)-pattern enhanced ILU(p, q) method, the re-ordering have to performed only on the decomposed-color blocks.

3.7 Approximate Inverse Preconditioners

As an explicit type of preconditioner, the approximate inverse requires only a matrix-vector multiplication in order to solve the preconditioning step. In this respect this kind of preconditioner is naturally parallel.

3.8 Other Parallel Preconditioning Techniques

3.8.1 Support-tree/Vaidya Preconditioners

Support-tree/Vaidya preconditioners provide parallel solving phase of the preconditioning equation [23, 25, 26, 27, 28]. However they are limited to diagonally dominant symmetric matrices. Our goal is to provide more generic techniques. Nevertheless, we believe that for some symmetric and diagonally dominant matrices these preconditioners can provide efficient and parallel schemes suitable for the multi-core and many-core platforms.

3.8.2 Schur Complement Preconditioners

In contrast to the block Jacobi/additive Schwarz preconditioners, the Schur complement handles also the couplings with the other sub-blocks of the problem. The size of the skeleton-matrix (the matrix providing the couplings of the sub-blocks) grows with the number of sub-blocks. This leads to an unbalanced level of parallelism. Thus, on homogeneous parallel devices this algorithm does not provide uniform load to all compute units. However, heterogeneous platforms could benefit from this scheme if the skeleton-system can be efficiently solved on them, which results in a problem-specific solution procedure.

3.8.3 Hierarchical Matrix Preconditioners

Preconditioners based on the approximate inverse with hierarchical matrices as well as standard approximate inverse preconditioners can be applied efficiently in parallel, see [17, 32, 55]. The LU-decomposed preconditioners with hierarchical matrices are processed with triangular solvers, which cannot be executed in parallel and thus are not suited for parallel devices.

3.8.4 Tridiagonal Preconditioners

Parallel algorithms for solving tridiagonal systems based on cyclic reduction and recursive doubling are presented in [48]. These parallel solvers have logarithmic complexity in comparison to their sequential versions which have linear complexity. In the general case of solving non-tridiagonal systems, we can use a tridiagonal approximation to the matrix as a preconditioner. Thus, we can consider it as a part of the splitting preconditioner (e.g. Jacobi and Gauss-Seidel).

But as such, this scheme has a limited impact as preconditioner. However, [48] showed that the line-wise smoother has a good smoothing properties for locally structured matrices in the context of multi-grid solvers.

3.8.5 Iterative Schemes as (Non-)Constant Preconditioners

It is not common practice to use standard iterative solvers (e.g. Gauss-Seidel iterations) as preconditioners – this is mainly because although we decrease the number of iterations of the outer solver, the total computation time typically grows. However, there are some cases when the inner solver (the preconditioning solver) is fast but cannot provide high accuracy. Thus, it can be used as an acceleration technique – examples for that are multi-grid methods, Chebyshev-iteration and low precision solvers combined with a defect correction step. In many cases, due to the large number of parameters, the multi-grid solver can be used less tuned as a preconditioner. An implementation of a multi-grid solver in a BiCGStab can be found in [48]. If the spectrum of the matirx A is known, then a Chebyshev-iteration can be used as an acceleration scheme. In some cases, it is possible to represent the matrix in a low precision format (which is typically faster to compute) and via defect correction scheme to use the lower precision system as a preconditioner. The main problem here is to determine when it is possible to transfer the problem into a lower precision representation. Related results on GPU devices can be found in [6, 7, 8, 110].

Chapter 4

Local Multi-Platform Linear Algebra Toolbox in HiFlow^3

In this chapter, we present our concept and implementation of the multi-platform linear algebra toolbox for sparse iterative solvers. We give a brief introduction to the current microprocessor technologies. We make an overview of the memory access model and the key object methods that help us to build efficient preconditioners and solvers for various platforms. We also explain how this design helps us to maintain a single source code program.

4.1 Emerging Multi-core and Many-core Devices

In this section, we give a brief overview of the current microprocessor technologies focusing on Central Processing Units (CPUs) and Graphical Processing Units (GPUs). In addition, we provide a basic information about the programming models and languages specifications for these devices.

4.1.1 Central Processing Unit

The CPU is capable to perform not only general purpose operations (e.g. reading/writing from a hard disk, accessing an ethernet device, etc) but also floating-point operations. Modern CPU devices contain several cores (typically 2 to 12) which share different levels of cache. Several processors can be combined on a single motherboard that has a common address space. The memory system with respect to the organization can be uniform, called Uniform Memory Access (UMA), or non-uniform, called Non-uniform Memory Access (NUMA). In the first case, the memory is uniformly accessible from all CPUs. In the second case, each CPU has an attached memory which can also be accessed from the other CPUs. This results in different bandwidth and latency when accessing local or remote memory.

One of the most popular programming models for programming multi-core devices is the fork-join approach. With this technique, a loop-based section can be accelerated by splitting the work into several partial loops and performing them in parallel. For a shared memory system, the most common programming language is Open Multi-Processing (OpenMP) [106]. Based on it, we can declare a parallel section where the compiler takes care of the proper spawn and control of the threads. OpenMP is in contrast to the Pthreads [68], where the programmer is responsible for launching and controlling the threads.

4.1.2 Graphics Processing Unit

Accelerator boards have been used for a long time to speed up the computation of many scientific applications. In the last few years, GPU devices are also making an impact in this sector.

Originally, for graphics visualization the GPUs were designed to perform a large amount of integer operations in parallel. But now GPU devices are capable to perform integers and float-point (single and double precision) operations. Thus, these devices have been transformed to a General Purpose platforms (GP-GPU). They are highly-parallel (containing hundreds of cores) and designed to work with stream programming models. In the NVIDIA cards, each processing unit consists of eight cores [100]. All threads in this processing unit can access small and fast memory called shared-memory. This memory is manual, i.e. the programmer is responsible for copying data from the global memory to the shared-memory in order to use it. Additionally, NVIDIA GPUs also provide texture memory which can be used as a read-only cache. This memory is very small – only few KBytes which is in contrast to the CPUs where caches are of order of MBytes.

The GPUs are based on stream programming model since they are highly-parallel devices without explicit communication among the compute units. These architecture and programming model are very different from the well-established CPU models using sequential or fork-join parallel techniques. The stream model is based on data-parallelism and does not provide global synchronizations or communication mechanisms in the threads. However, some languages (as CUDA [100] and OpenCL [78]), provide locally barrier mechanism which works for a group of threads. The most remarkable feature of this model is that it is naturally parallel.

4.1.3 Upcoming Technologies

The next generation of microprocessor technology is hard to predict. However, it is reasonable to generalize that processors designed for high-performance computing tend to have large amount of cores and small local memory [9]. This can be seen from the announced devices as Intel MIC [72] and Intel SCC [74].

In the future of parallel programming models, there remain many open questions about memory transaction, locality and recursive calls, but one thing is certain – the models have to be scalable. Following the work of [96], we believe that future programming models will include some form of stream base models with extensions for specific hardware architectures.

4.2 HiFlow^3

HiFlow^3 is a multi-purpose finite element package for solving wide range of problems modeled by PDEs, see [4, 64]. It is based on the mesh, Degree of Freedom/FEM (DOF/FEM) and Linear Algebra Toolbox (LAToolbox) modules. The software stack is written in C++ and it provides modular techniques which give high flexibility. In order to solve large scale problems efficiently, the software stack has been designed to be highly-parallel. Special focus is the hardware-aware computation of the solvers in HiFlow^3. The LAToolbox module handles the basic linear algebra operations and offers linear solvers and preconditioners. It is implemented as a two-level library. The global level is an MPI-layer which handles the distribution of the data among the nodes and performs cross-node computations. The local level (local multi-platform LAToolbox) takes care of the on-node routines, offering an interface independent of the specific platform. It supports different types of parallel platforms and handles different programming models for using them. In this section we present its concepts, structure and implementation aspects.

4.3 Goal and Design

The goal of the local multi-platform Linear Algebra Toolbox (lmpLAToolbox) is to provide a complete set of generic and portable routines and/or interfaces to them for different parallel platforms.

The core of the module are two classes - vectors and matrices. The vector object contains the vector data and provides all vector routines/interfaces (e.g. vector update, scaling, rotation, scalar product, etc). Similarly, the matrix object contains the matrix data in sparse format and it provides all matrix routines (matrix-vector multiplication, matrix-matrix multiplication, scaling functions and etc).

4.3.1 Abstraction

The main concept for this library is based on abstraction. As in the object-oriented principle, each vector/matrix object can be manipulated only via its interface. This interface contains all the necessary vector-vector, matrix-vector and matrix-matrix operations. The solvers and preconditioners are built using only the interface with these objects and so it is abstracted from any particular platform or implementation. Every platform can be added by just contributing the specific backend and providing all matrix and vector routines via the defined interface.

The implementation of this abstraction is to ensure portability of the solver over the variety of different parallel platforms and parallel languages. The solvers are independent of any particular implementation, platform or format representation. Even more, the matrix format is irrelevant of the solving of the preconditioning phase. This is possible due to the way in which the preconditioning phase is performed, see Section 3.4. In contrast to that, typically, a special matrix format (e.g. CSR) is necessary for the forward and backward substitutions in the LU-based preconditioners, see [73, 99, 114].

4.3.2 New Hardware/Languages and Development Cycles

The driving force for developing this library is the ability to use it and maintain it over the next generations of parallel many-core devices. Our experience shows that the development cycle of a complex and sophisticated numerical scheme and its implementation is much longer than the production cycle of parallel microprocessors. If not designed properly, the implementation of a numerical solver will only run efficiently and be fully utilized on specific hardware. By trying to perform our algorithm in parallel on a suitable system, we face also a variety of different paradigms and languages for programming these hardware.

Therefore, our aim is to provide a generic, portable and flexible framework for performing preconditioned sparse iterative linear solvers with a good efficiency and scalability.

4.4 Structure of lmpLAToolbox

In Figure 4.1 the structure of the hierarchy in the lmpLAToolbox is presented. Each matrix/vector class has its base class, followed by a platform-management class which takes care of the memory allocation, placement and access for all the available platforms. The platform-management class (called data management level) inherits its interface from the base class. It is responsible for the data transfers among the different backends (e.g. CPU-GPU and GPU-CPU communication). Each particular and platform-specific implementation is inherited from the data management class. The base class for each vector and matrix object provides complete interface for all routines and operations. Typically, the lmpLAToolbox is used only by the interface of the pointers to the base classes. Detailed information can be found in [4, 57], an example of a preconditioned CG solver is presented in Appendix A. In this way we do not have to specify any particular platform – the decision for the platform and implementation can be taken at run-time.

Via the base class interface, each matrix or vector object can perform one of the following:

- Execute vector/matrix operations with itself or objects on the same platform (not necessarily with the same implementation).

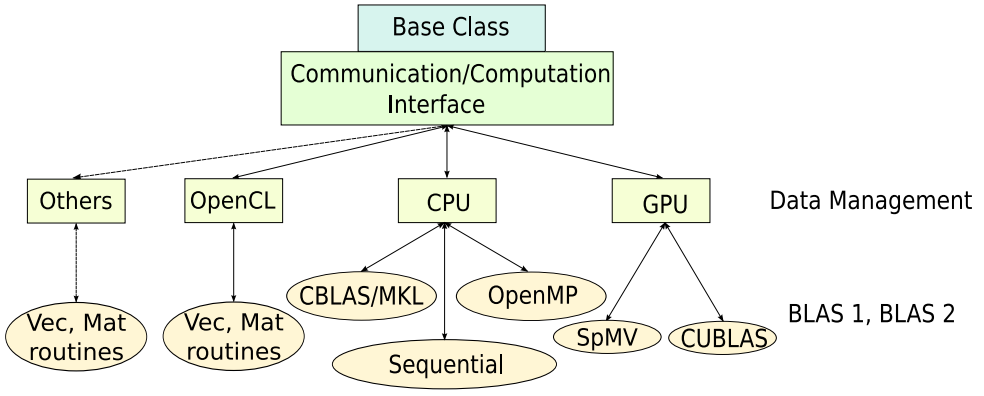

Figure 4.1: The local multi-platform linear algebra toolbox (lmpLAToolbox)

- Clone itself, e.g. create an object based on the same platform and implementation.
- Copy its data (matrix/vector) to a similar or different object on a different platform and/or implementation.
- Use advanced data techniques such as vector splitting, vector concatenating, or data extraction based on irregular patterns.
- Call routines for pre-processing steps such as I/O to files, incomplete LU factorization, graph analysis, permutation and assembling routines on the CPU classes.

The current version of the library is based on (but not limited to) the CSR matrix format, see [15, 114]. This format has been chosen because of the easy access in the pre-processing steps (e.g. graph analyzing and decomposition routines) and because of its good performance behavior for general sparse matrices. The symmetric matrices are stored and handled as in the general non-symmetric case. The library supports also zero size vectors and matrices (i.e. empty vectors/matrices) which are mainly used in the preconditioners.

4.4.1 Memory Access

By using only pointers to the base class and working only with its interface, the user has no raw access to the data of the objects. Direct access to the platform data can be obtained by the platform specific classes. Thus we need to make an explicit declaration of a particular type of a vector or a matrix.

For periodic irregular memory accesses (e.g. exchange of the ghost values, see Section 4.6), the library provides a special index set in the vector classes. With this index set, the user can specify a particular set of values (non-contiguous in memory) of the vector and then by a single function the values can be extracted in a CPU buffer. This technique provides special buffering mechanisms which is very powerful for devices attached via high-latency bus (e.g. GPU devices on PCI bus), see [57]. An example is consider later in Chapter 5

Operations such as assembling of the stiffness matrix and the right-hand side (see (2.10) and (2.11)) can be done in the CPU-based classes via the global interface.

4.4.2 Different Backends

The lmpLAToolbox can be seen as an interface to specific backends. The major task of the library is to take care of the memory management between different platform. Furthermore, it provides implementations of the vector-vector and sparse matrix-vector routines for several platforms. The rest of the functions are interface to libraries (e.g. ATLAS [10], CUBLAS [101], etc).

4.4.2.1 CPU

This is the largest backend with respect to functionality. Here, we provide a sequential version and an OpenMP parallel implementation of the major part of the BLAS routines. This backend has extra functions which are designed for pre-processing steps: I/O to files, ILU factorization, graph analysis (e.g. multi-coloring, level-scheduling, (reverse) Cuthill-McKee ordering [38]) and permutation routines. The CPU-backend also provides assembling routines for building sparse matrices and vectors. Furthermore, the library provides interface to the ATLAS/MKL library, see [10, 73].

4.4.2.2 GPU/NVIDIA

The NVIDIA GPU backed is based on the CUDA programming language, see [100]. The vector routines in the lmpLAToolbox are executed via the CUBLAS library (see [101]) and the sparse matrix-vector routines are implemented accordingly to [16, 19]. The backend provides tuning features like specific device allocation, texture caching and block sizes.

4.4.2.3 OpenCL

With the OpenCL (see [78]) backend we target a larger variety of parallel platforms – current CPU/GPU platforms and future systems compatible with a stream based model. However, for many platforms the library provides platform-specific optimized routines for better performance – reduction based routines are implemented in different ways for ATI GPUs, NVIDIA GPUs and Intel/AMD CPUs.

4.4.2.4 Others

As mentioned before, the library is open for further extensions. Currently, for in-house use, we have developed a multi-precision backend based on the GNU MP library [47].

4.5 Portable Linear Solvers and Preconditioners

Due to the flexibility of the library and the way it can be used, we can build full-parallel schemes (platform-independent) or hybrid-parallel schemes (with partially platform specific routines).

4.5.1 Full-parallel Schemes

The full-parallel solvers provided by the library are multi-grid, CG and BiCGStab. An example of a CG solver is presented in Appendix A, see [57]. However, the developer is free to develop his/her own full-parallel algorithms. The preconditioning step is one of the most frequently executed routines. In order to provide good scalability, the library handles all the preconditioners (i.e. additive, multiplicative based on level-scheduling or power(q)-pattern method, and approximate inverse) in a full-parallel manner. The library supports classical additive methods such as Jacobi, (symmetric) Gauss-Seidel, (symmetric) SOR based on multi-coloring decomposition; multiplicative methods such as incomplete LU decomposition based on level-scheduling or power(q)-pattern method; approximate inverse – FSAI method. For these preconditioners, the solving phases are considered in Section 3.4. As an example, in Algorithm 9 we present pseudo-code of the parallel sweeps for multi-colored symmetric Gauss-Seidel preconditioners. Benchmarks are presented in Chapter 5.

Algorithm 9 LU-sweeps for solving $Mz = r$ for symmetric Gauss-Seidel

Split and copy vector r into z_i $i = 1, ..., B$, where B = numcolor or numlevels

Forward step $z_i := D_i^{-1}(z_i - \sum_{j=1}^{i-1} L_{i,j} z_j)$ for $i = 1, \ldots, B$

Diagonal step $z_i := D_i z_i$ for $i = 1, \ldots, B$

Backward step $z_i := D_i^{-1}(z_i - \sum_{j=1}^{B-i} R_{i,j} z_{i+j})$ for $i = 1, \ldots, B$

Concatenate vector z_i into z

4.5.2 Hybrid-parallel Schemes

The library also allows us to develop hybrid-parallel schemes – a typical example of this is the implementation of the GMRES solver. All the vector-vector and matrix-vector routines can be executed in parallel. But due to the computation of the Hessenberg matrix the platform performs all of the sequential steps on the CPU, see [58, 114]. Numerical experiments and benchmarks are considered in Chapter 5.

4.6 lmpLAToolbox on Heterogeneous HPC Clusters

The LAToolbox in HiFlow^3 is a two-level library – a global MPI-layer which handles computation and communication over the nodes, and a local layer which performs the actual vector or matrix routines (lmpLAToolbox). The LAToolbox in HiFlow^3 is an example, of the way the local multi-platform toolbox can be used in a global environment context. In this case, the global layer manages the global scatter and gather operations – reduction-based routines and sparse matrix-vector multiplications, see [4, 57].

A typical domain distribution is presented in Figure 4.2. To perform the SpMV on the global level, we need to perform a local SpMV and to add the contributing couplings from the neighboring domains. In this example, the process P_0 owns the interior nodes (nodes locally for this process) and it needs to obtain the ghost nodes (nodes which do not belong to this process) from the other processes (P_1, P_2 and P_3). In the figure, we denote the interior of the blocks (diagonal elements) with green. With blue, violet and red colors are denoted the ghost layers (off-diagonal elements) which need to be exchanged during the SpMV.

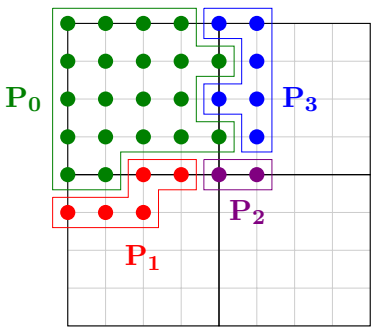

Figure 4.2: Example of a domain partitioning of 4 blocks – with green we denote the DOF of process P_0 and the remaining DOF represent the inter-process couplings for process P_0

The algebraic procedure for computing the multiplication can be split into two parts – computation of the diagonal block and computation of the contributions from the off-diagonal parts, see Figure 4.3. To minimize the computation time, we overlap the communication step with the computation of the diagonal block, see Algorithm 10. As we have discussed above, the lmpLA-Toolbox provides special global functions for extracting these values disregarding the platform

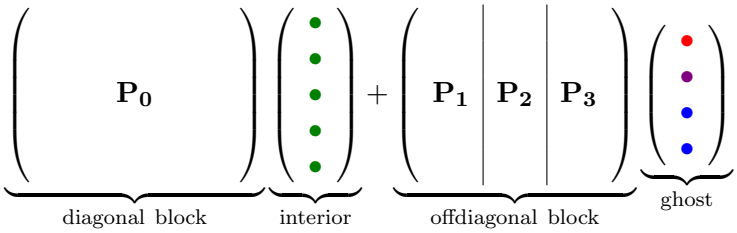

Figure 4.3: Distributed matrix-vector multiplication

Algorithm 10 Distributed matrix vector multiplication $y = Ax$

Start asynchronous communication – exchanging ghost values;
$y_{\text{int}} = A_{\text{diag}}\, x_{\text{int}}$;
Synchronize communication;
$y_{\text{int}} = y_{\text{int}} + A_{\text{offdiag}}\, x_{\text{ghost}}$;

choice. Thus, we can even combine different backends on a cluster. Performance benchmarks are presented in Chapter 5, details can also be found in [4, 58, 57].

Chapter 5

Numerical Experiments and Performance Analysis

In this chapter, we give a qualitative performance analysis of parallel linear solvers and preconditioners on a variety of problems. First, we consider the Poisson and convection-diffusion problems. For the Poisson problem we present performance results for the preconditioned CG method and for the multi-grid solver on a 2D L-shaped domain with a locally refined grid. The convection-diffusion equation is solved with preconditioned GMRES. We present results that demonstrate the impact of the physical dimensions (2D, 3D) and choice of the finite element spaces (linear, quadratic) on the multi-coloring decomposition. Furthermore, after factorization, we investigate the sparsity patterns for the level-scheduling and power(q)-pattern method. We show the execution behavior of the preconditioned solvers applied to different matrix problems. These tests show the strength of the power(q)-pattern method and the ability to use it as an out-of-the-box scheme. For the preconditioned solvers we present the impact on the number of iterations as well as on the total computational time on multi-core CPU and GPU devices.

5.1 Problems Description

5.1.1 Poisson Equation

The Poisson equation is a second order elliptic PDE of the form

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_D \quad \text{on } \partial\Omega, \end{aligned} \tag{5.1}$$

where $\Omega \subseteq \mathbb{R}^d$ is a domain, $d = 1, 2, 3$ and Δ is the Laplace operator.

For a physical interpretation of this equation we refer to the potential theory [121]. As an example, we can consider u as a function which represents the electrostatic potential field. The field is induced by a charge distribution of a density function f.

We solve the Poisson problem with homogeneous Dirichlet boundary conditions $u_D = 0$ and prescribe the right hand side $f = 8\pi^2 \cos(2\pi x)\cos(2\pi y)$. In this experiment Ω is a 2D L-shaped domain. Due to the steep gradient of the solution close to the re-entrant corner, we use locally refined meshes in order to obtain a proper resolution. A plot of the solution and the used mesh is presented in Figure 5.1. To underline the necessity we present a plot of the gradient of the solution with a globally refined mesh and with a locally refined mesh. Clearly, with the same amount of unknowns we can represent the solution of the problem with higher accuracy only with a locally refined mesh, see Figure 5.2. The mesh for the 2D L-shaped domain is obtained by refining the quadrilateral elements close to the edge where we expect to obtain a steep gradient of the solution. The resulting statically adapted grid for this 2D L-shaped domain contains

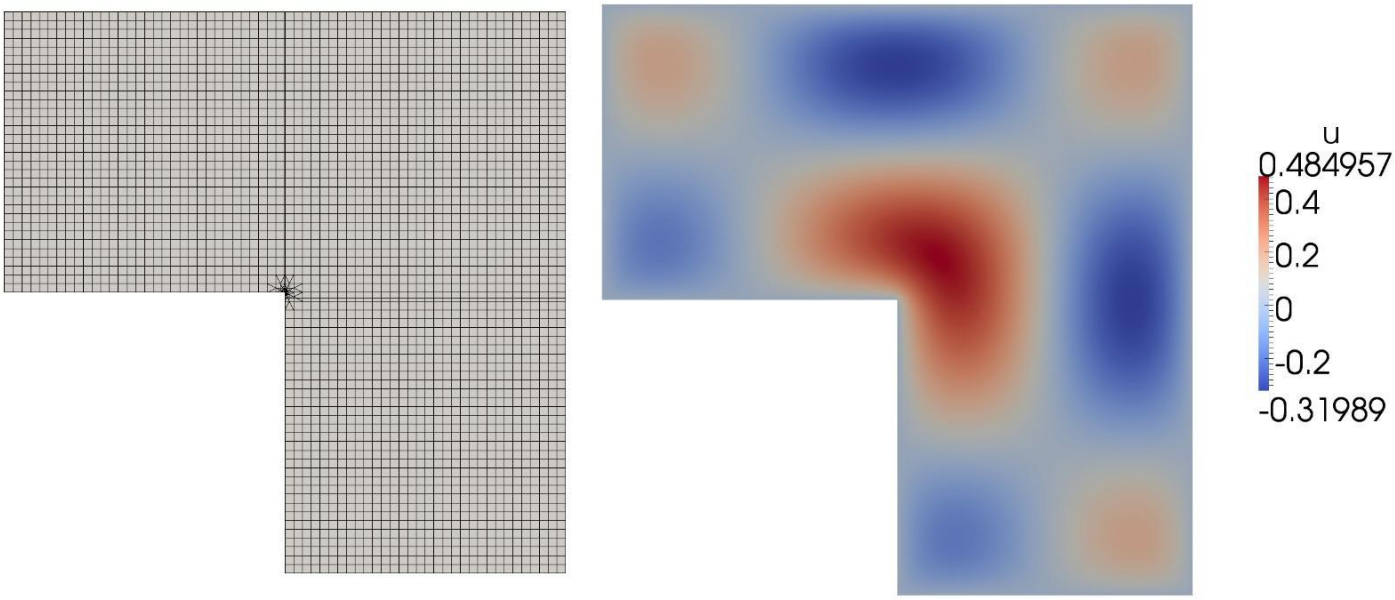

Figure 5.1: Locally refined L-shaped domain (left) and the corresponding solution of the Poisson problem (right)

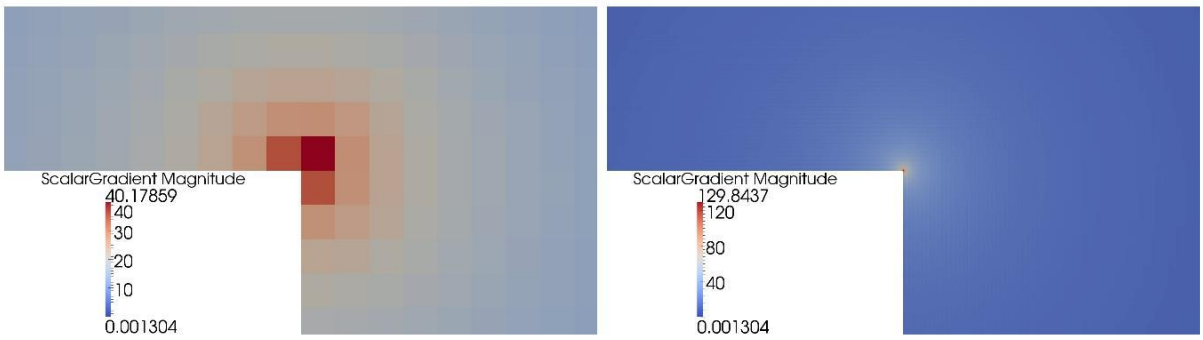

Figure 5.2: Zoomed-in plots of the gradient of the Poisson solution on a uniform mesh with 3,149,825 DOF (left) and on a locally refined mesh with 3,211,425 DOF (right) – our finest multi-grid mesh with level 6

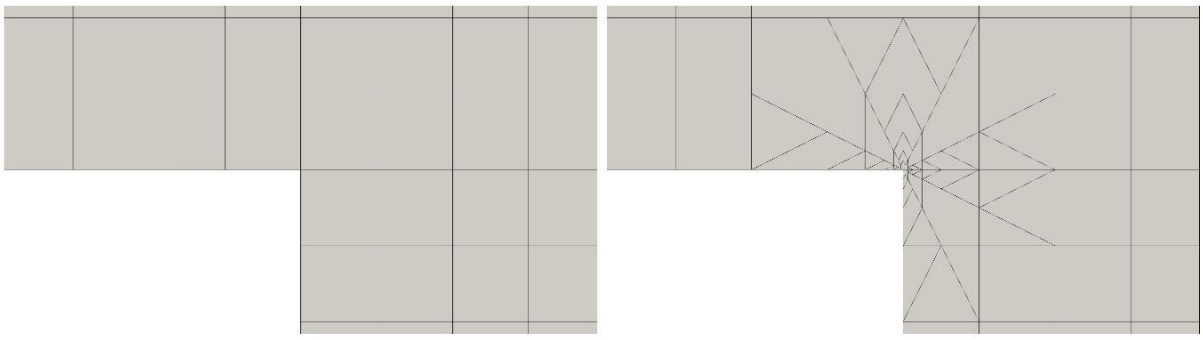

Figure 5.3: Zoomed-in grids based on a uniform mesh with 3,201 DOF (left) and a locally refined mesh with 3,266 DOF (right) of the L-shaped domain around the re-entrant corner – our coarsest multi-grid mesh with level 1

quadrilateral and triangular cells. By introducing these triangular cells, we avoid the hanging nodes that occur when we apply a local refinement, see Figure 5.3.

We solve the problem by means of FEM based on P1 (2.12) and Q1 (2.13) elements. The

coarsest mesh contains 3,266 degrees of freedom. By applying six global refinements we obtain our finest mesh resulting in 3,211,425 degrees of freedom. Table 5.1 presents the characteristics of the L-shaped domain with respect to the number of cells, degrees of freedom and discretization steps.

# Level	#Cells	#DOF	$h_{\max}$	$h_{\min}$
1	3,177	3,266	0.015625	0.000488
2	12,708	12,795	0.0078125	0.000244
3	50,832	50,645	0.00390625	0.000122
4	203,328	201,513	0.001953125	0.0000610
5	813,312	803,921	0.000976563	0.0000305
6	3,253,248	3,211,425	0.000488281	0.00001525

Table 5.1: Characteristics of the six refinement levels of the locally refined L-shaped domain

5.1.2 Convection-diffusion Equation

As another model problem, we use the convection-diffusion equation. It typically describes slow diffusion and fast transport processes: the distribution of the temperature in compressible flows, the concentration of pollutants in fluids, and the momentum relation in the Navier-Stokes equations are modeled by means of convection-diffusion processes. In many scenarios, the convection part dominates the diffusion part. Consequently, smoothing properties of the second order Laplacian-like diffusion operator are obfuscated. In the stationary case, neglecting reaction terms and assuming constant coefficients $(\mathbf{b}, \varepsilon)$, the considered two and three dimensional $(d = 2, 3)$ convection-diffusion equation is elliptic and reads

$$\begin{aligned} -\varepsilon \Delta u + \mathbf{b} \cdot \nabla u &= f \quad \text{in } \Omega, \\ u &= u_D \quad \text{on } \partial\Omega. \end{aligned} \tag{5.2}$$

Here, $\Omega \subseteq \mathbb{R}^d$ is a bounded domain and $\mathbf{b} = (K_1, \ldots, K_d)$ represents the directions of the transport and the information flow. For small diffusion coefficients, i.e. $\|\mathbf{b}\| \gg \varepsilon$, the solution typically shows layers at the boundary of the domain, see [111].

We solve (5.2) numerically by means of FEM. We choose a characteristic mesh size h which results in a number of mesh points of order $O(h^{-d})$. In order to prevent numerical oscillations, the stability constraint

$$h \leq C \frac{\varepsilon}{\|\mathbf{b}\|}$$

for a given constant C needs to be imposed [111]. In these tests, we use parameters $K_1 = K_2 (= K_3) = 120$ and $\varepsilon = 1.0$ and we impose zero Dirichlet boundary conditions, i.e. $u_D = 0$.

For our numerical experiments, we consider exact solutions given by

$$u(x, y) = x \frac{e^{K_1 x} - e^{K_1}}{e^{K_1}} \sin(\pi y), \qquad (x, y) \in \Omega := [0, 1]^2, \tag{5.3}$$

$$u(x, y, z) = x \frac{e^{K_1 x} - e^{K_1}}{e^{K_1}} \sin(\pi y) \sin(\pi z), \quad (x, y, z) \in \Omega := [0, 1]^3, \tag{5.4}$$

with right-hand sides prescribed by inserting these solutions into (5.2). The solutions are plotted in Figure 5.4. The boundary layer formed by the difference in the exponents close to $x = 1.0$ can be seen in both of the figures.

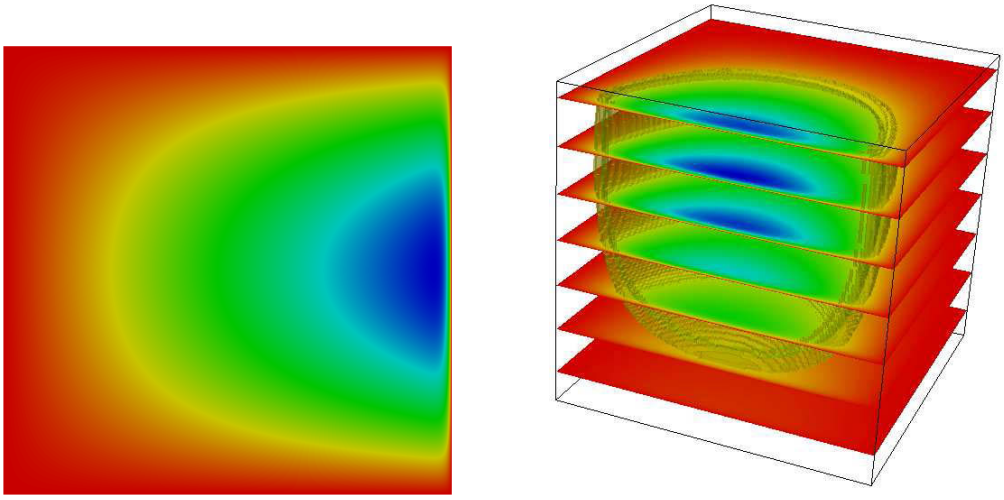

Figure 5.4: Solutions of the convection-diffusion problem given in (5.3) and (5.4) in the 2D (left) and 3D case (right)

Name	Description of the problem	#rows	#non-zeroes	#colors	#block-SpMV in ILU(0,1)
`ecology2`	Animal/gene movement	999,999	4,995,991	2	2
`s3dkq4m2`	Cylindrical shells	90,449	4,820,891	24	552
`g3_circuit`	Circuit simulation	1,585,478	7,660,826	4	12

Table 5.2: Description and properties of the considered test matrices

5.1.3 Linear System Problems Based on Matrix Collection

The matrix test suite for performance evaluation is based on three real-valued symmetric and positive definite matrices from three different application areas. The `ecology2` matrix is derived from a landscape ecology problem based on electrical network theory for modeling of a 2D animal/gene movement flow [125]. The `s3dkq4m2` matrix is obtained from a finite element analysis of cylindrical shells on a uniform quadrilateral mesh [95]. The `g3_circuit` matrix results from a circuit simulation problem [126]. Table 5.2 lists the properties of the test matrices. It shows the number of rows, columns and colors when the multi-coloring method is applied to the original matrix. In the last column, the number of sparse matrix-vector operations with respect to the block decomposition for the multi-colored ILU(0,1) preconditioner is given. For the `g3_circuit` matrix the decomposition into colors (by applying the multi-coloring permutation to the original matrix) is imbalanced – we get 689,390, 789,436, 106,502 and 150 entries per different color. For the `s3dkq4m2` and `ecology2` matrix the block distributions have balanced sizes. The average number of non-zero elements per row is 4.9, 53.29 and 4.83 for `ecology2`, `s3dkq4m2` and `g3_circuit` respectively.

In order to present the sparsity pattern behavior after the ILU factorization process, we use two small matrices. The symmetric and positive definite `nos5` matrix of size 468-by-468 and 5,172 non-zero elements describes a finite element approximation of beams with one free and one fixed end [86]. The symmetric and positive definite `gr3030` matrix is derived by a finite difference discretization of a Laplace problem. The problem has dimension 900-by-900 with 7,744 non-zero entries [91]. The original sparsity patterns are shown in Figure 5.5 with `nos5` on the left and `gr3030` on the right.

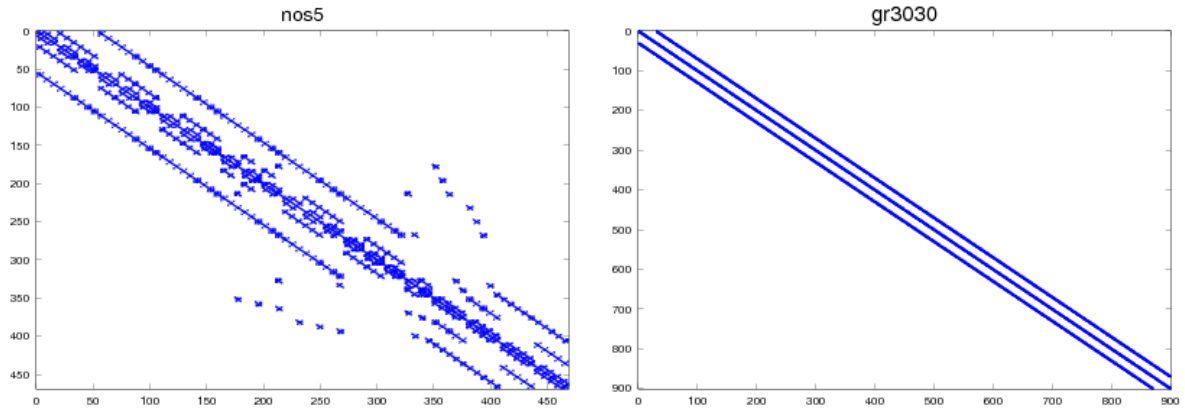

Figure 5.5: Sparsity patterns of the `nos5` matrix of size 468-by-468 with 5,172 non-zero elements (left) and the `gr3030` matrix of size 900-by-900 with 7,744 non-zero elements (right)

5.2 Solution Procedure

5.2.1 Conjugate Gradient Solver

To obtain the solution of the Poisson problem (5.1) and the problems based on the symmetric and positive definite matrices defined in Section 5.1.3, we use the CG method. The solver is fully implemented in parallel without platform-specific or sequential parts, see Chapter 4. We use zero initial values and relative stopping criterion of 10^{-6} for the residual. For all of the tests in the matrix collection we set the components of the right-hand side to one.

5.2.2 Generalized Minimal Residual Solver

For the convection-diffusion problem (5.2) we use a GMRES solver. Except the computation of the Hessenberg matrix which is done strictly on the CPU device, all of the vector-vector and matrix-vector routines are fully executed in parallel. To control the memory usage, we use a restarted GMRES with the size of the Kyrlov subspace 30, see [114]. We prescribe a relative or an absolute stopping criteria for the residual of 10^{-19} and 10^{-11} respectively.

5.2.3 Multi-grid Solver

We consider matrix-based geometric multi-grid methods [117]. In contrast to the stencil-based geometric multi-grid methods, all differential operators, smoothers and inter-grid transfer operators are not expressed by fixed stencils on equidistant grids but have the full flexibility of sparse matrix representations. This approach gives flexibility with respect to complex geometries and non-uniform grids resulting from local mesh refinements and space-dependent coefficients in the underlying PDE. Moreover, as implemented in the LAToolbox in HiFlow^3, the solvers can be built on standard building blocks contained in numerical libraries.

All routines of the multi-grid solver are performed in parallel, including the solution of the coarse system which is solved with the CG method. For the inter-grid transfer operations we use the full-weighted restrictions and the bi-linear interpolations based on the topology information of the underlying grid. The preconditioned defect correction scheme (2.20) combined with different preconditioning matrices is used as a smoother. The differential operator, as well as the inter-grid transfer operators, are built as sparse matrices and are applied using a parallel SpMV. Thus, we assemble all of the matrices in the pre-processing step of the solver.

5.2.4 Preconditioners

5.2.4.1 Additive Preconditioners

For the additive preconditioners we use the multi-coloring decomposition algorithm to obtain parallelism. To accelerate the Jacobi preconditioner and to avoid matrix indexing, we explicitly extract the diagonal of the matrix and create a vector out of it. Thus, we can apply the Jacobi preconditioner by a component-wise vector-vector multiplication.

5.2.4.2 Multiplicative Preconditioners

For the multiplicative preconditioners we use ILU decompositions. To perform the forward and backward substitutions in parallel, we use the level-scheduling method (see Section 3.6.7) and the power(q)-pattern method (see Section 3.6.8). In general, the permutations obtained by the level-scheduling algorithm are different for the L and for the U matrix. Typically, more elements are obtained in the L part of the matrix. However, for all the symmetric matrices in our test suite they are either the same or $\mathcal{NNZ}(L) \supseteq \mathcal{NNZ}(U)$ and we can use a permutation based only on L. Thus, in our test cases, for both level-scheduling and power(q)-pattern method we apply the permutation before performing the iterative solver.

Even for the symmetric and positive definite matrices we use a LU and not (incomplete) Cholesky decomposition. The reason for doing this is that we need to perform the block-sweeps (3.6) and (3.7). This involves a multiplication with the transposed sub-matrices $L_{i,j}^T$. Due to the sparse data format, this is an expensive parallel operation. Thus, we need to store the transposed matrices in addition which lead to the same memory requirements as for the LU decomposition.

5.2.4.3 Approximate Inverse Preconditioners

By explicitly building the inverse of the preconditioning matrix, we need to apply only a parallel SpMV operation in the preconditioning solution phase. We use the FSAI algorithm to compute the matrix, see Section 2.5.3. As in the pattern control of the Chebyshev polynomial preconditioners (see (2.23) and (2.22)), we build our sparsity pattern in a similar way. With FSAI_q we denote the produced matrix built with its prescribed sparsity pattern of $|A|^q$. Due to the expensive parallel multiplication of the transposed matrix, we keep both the lower-triangular matrix and its transposed one, see formula (2.26).

5.3 Behavior of the Parallel Linear Solvers

In this section, we present the behavior of the preconditioned linear solvers with respect to their performance efficiency in terms of number of iterations. For the parallel additive, multiplicative and approximate inverse preconditioners we present their acceleration factors in terms of reduction of the iteration counts. Furthermore, we show the sparsity patterns and degree of parallelism for the level-scheduling and the power(q)-pattern methods.

5.3.1 Impact on Multi-coloring Decomposition

We investigate the impact of the dimension d and the choice of the finite element space on the multi-coloring decomposition used in the additive and ILU(0,1) preconditioners. For this we study the convection-diffusion equation (5.2) and apply standard finite element discretization based on Q1 or Q2 Lagrangian elements [52, 30]. This leads to linear systems with a non-symmetric matrices in which the transport terms are related to the non-symmetric contributions. The condition numbers of the discretized problems depend on several parameters such as mesh characteristics and finite element spaces, and it increases polynomially in h^{-2} [111]. The used number of unknowns for discretization by means of Q1 and Q2 elements in two and three

dimensions and the corresponding number of non-zero elements in the sparse system matrices are presented in Table 5.3.

	Q1 elements			Q2 elements		
	#Unknowns	#Non-zeroes	#Colors	#Unknowns	#Non-zeroes	#Colors
2D	1025^2	$9 \cdot 10^6$	5	513^2	$16 \cdot 10^6$	11
3D	129^3	$57 \cdot 10^6$	13	65^3	$135 \cdot 10^6$	35

Table 5.3: Number of used degrees of freedom and number of non-zero elements for the discretized convection-diffusion equation in 2D and 3D

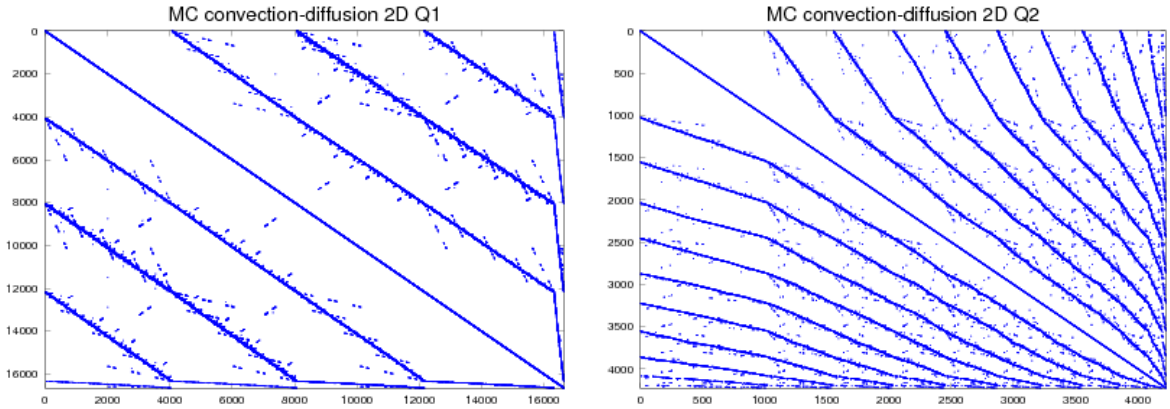

Figure 5.6: Non-zero pattern of the system matrices for the 2D discretization with multi-coloring permutation of the convection-diffusion equation based on Q1 (left) and Q2 (right) elements

The number of colors in a matrix obtained by finite element discretizations depends on the differential operator, the finite element test and trial functions (linear, quadratic or higher order), and the dimensions of the problem. The use of higher order elements leads to higher connectivity in the sparse matrix. Therefore, we observe an increased number of colors for the Q2 elements in comparison to the Q1 elements. A similar observation can be made for three-dimensional problems, see [58]. The structures of the matrices after re-ordering based on the multi-coloring decomposition for Q1 and Q2 elements in the two-dimensional case are depicted in Figure 5.6. The number of colors for Q1 finite elements is 5 and for Q2 elements is 11. In the three-dimensional case we find 13 colors for Q1 elements and 35 colors for Q2 elements, see Figure 5.7.

In Figure 5.8 we present the number of iterations for solving the convection-diffusion problem in two and three-dimensions with linear and quadratic finite elements. For preconditioning we use multi-colored Gauss-Seidel and ILU(0,1), and for the linear system we use GMRES. Furthermore, we investigate the performance of the block Jacobi-type preconditioner built on these two schemes.

5.3.2 Enhanced Parallel ILU(p)-based Preconditioners

Table 5.4 summarizes the number of iterations needed to achieve the prescribed error tolerance for the CG solver applied to the three test matrices, see [62]. The acceleration factor in terms of reduced number of iterations (#its), i.e #its(no precond)/#its(precond), goes up to 5.4 for the `ecology2` matrix, 554 for the `s3dkq4m2` matrix and 33 for the `g3_circuit` matrix. However, these numbers do not reflect the additional work and time consumed by the preconditioning step, i.e. the solution of the block-triangular systems. Moreover, Table 5.4 details the number of colors in the multi-colored SGS and ILU(0,1) scheme, in the power(q)-pattern enhanced multi-colored

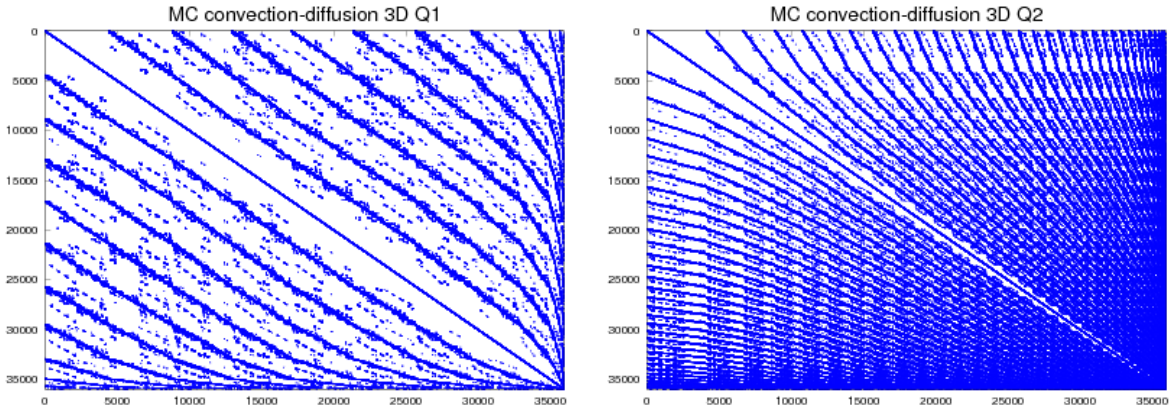

Figure 5.7: Non-zero pattern of the system matrices for the 3D discretization with multi-coloring permutation of the convection-diffusion equation based on Q1 (left) and Q2 (right) elements

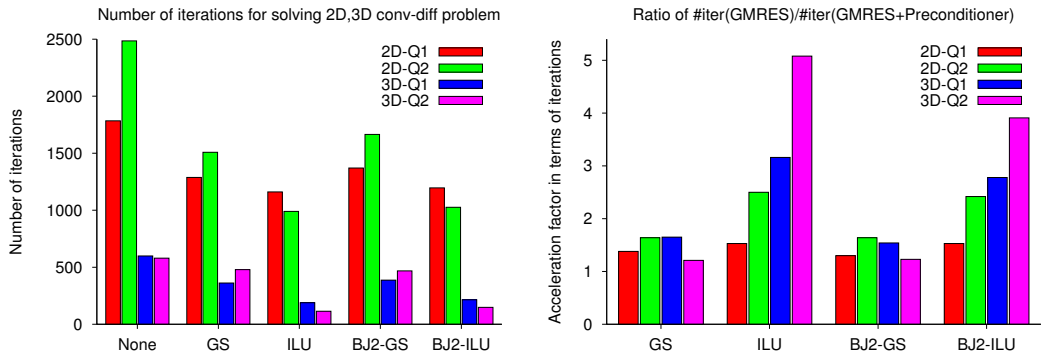

Figure 5.8: Number of iterations for solving the 2D/3D convection-diffusion problem based on Q1 and Q2 finite elements (left), and acceleration factors representing the ratio of the number of iterations of the non-preconditioned and of the preconditioned solver (right)

ILU(p,$p+1$) scheme, and in the drop-off version ILU(p,q) with $p = q = 3$. In comparison to these numbers, the number of levels in the level-scheduling algorithm applied to the original matrix is 2,593 for `ecology2`, 2,388 for `s3dkq4m2`, and 1,998 for `g3_circuit`.

5.3.3 Effect on Sparsity Pattern for Enhanced Parallel ILU(p)-based Preconditioners

In this section, we consider the impact of matrix re-ordering techniques on the structure and sparsity pattern of the system matrix, see [62]. The matrix decomposition into blocks with a lower bound on the block size is a necessary building block for fine-grained parallel methods. The number of elements per block determines the degree of parallelism while the number of blocks in the matrix decompositions is a measure for function call overheads (see the performance tests in Section 5.4.4 and Section 5.4.5).

By using small test matrices, we investigate how the multi-coloring, the level-scheduling and the power(q)-pattern enhanced multi-colored ILU(p,q) method determine the number of blocks (i.e. the number of colors) for the parallel execution and the sparsity pattern of the system matrix. In case of the ILU decomposition, the factorized matrices are also analyzed. Furthermore, we examine the impact of the choice of p and q in the power(q)-pattern enhanced ILU(p, q) decomposition with and without drop-off strategies. In this section, small test matrices are

		No precond	SGS	ILU(0,1)	ILU(1,2)	ILU(2,3)	ILU(3,4)	ILU(3,3)
ecology2	# its	5,391	2,783	2,855	1,815	1,308	997	1,277
	acc. fact.	1.0	1.93	1.88	2.90	4.12	5.40	4.22
	# colors		2	2	7	8	19	8
g3_circuit	#its	12,760	1,328	1,242	747	497	386	397
	acc. fact.	1.0	9.6	10.2	17.0	25.6	33.0	32.1
	# colors		4	4	10	17	35	17
s3dkq4m2	#its	535,056	12,728	3,918	2,203	1,600	965	6,086
	acc. fact.	1.0	42.0	136.5	242.8	334.4	554.4	87.9
	# colors		24	24	56	96	150	96

Table 5.4: Number of iterations of the preconditioned CG solver, acceleration factors with respect to reduced iteration count, and number of colors for the three test matrices

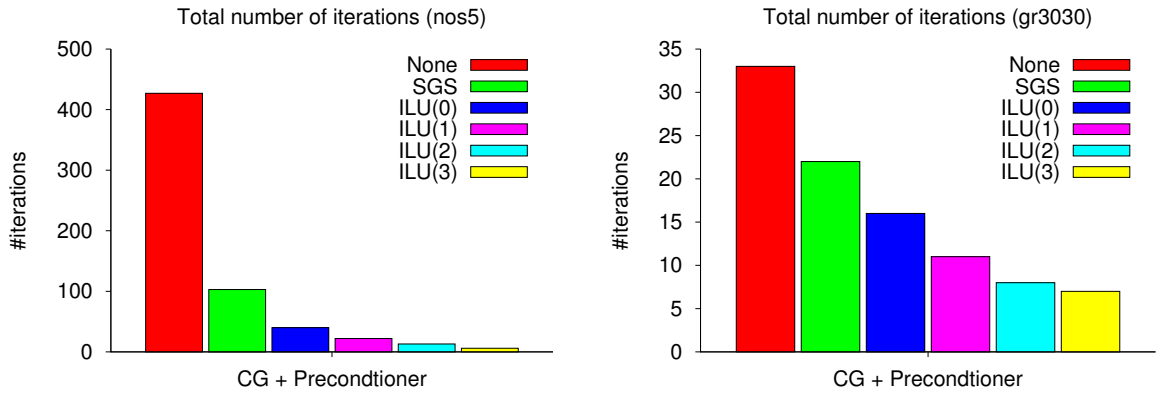

Figure 5.9: Number of CG iterations without preconditioner and with level-scheduling for SGS and ILU(p) preconditioners for $p = 0, 1, 2, 3$ for the `nos5` matrix (left) and the `gr3030` matrix (right)

chosen for a proper visualization of the matrix patterns in spy plots. Note that smaller matrices, such as the considered in this section, are not an appropriate input for fine-grained parallel preconditioners since they do not provide the necessary degree of parallelism due to their size. Larger matrices are considered later in this chapter.

Here, we consider the preconditioned CG as an iterative Krylov subspace-type solver. We set the right-hand side to one and use zero initial values. As shown in Figure 5.9 (left), for the `nos5` matrix more than 400 iterations are needed to achieve a relative residual smaller than 10^{-6}. With the multi-colored SGS preconditioner and the level-scheduling based ILU preconditioner with level-p fill-ins for $p = 0, 1, 2, 3$ the iteration count can be reduced to a factor 71. The ILU(p) efficiency with respect to the iteration count increases with p. A similar observation is made for the `gr3030` matrix in Figure 5.9 (right). The preconditioners decrease the number of iterations, ILU(3) has an acceleration factor of 4.7.

In Figure 5.10, the improvements of the power(q)-pattern enhanced multi-colored ILU(p,q) preconditioners (with and without drop-off) with respect to the CG iteration count are presented. We observe that the ILU(p,$p+1$) strategy without drop-off gives the best results in terms of reduction of the iteration count. These results also improve when increasing p. For the results with drop-off, i.e. with $q < p + 1$, the efficiency is slightly worse.

The sparsity patterns of $|A|^q$, $q = 1, 2, 3, 4$, for both matrices are illustrated in Figure 5.11 and Figure 5.12.

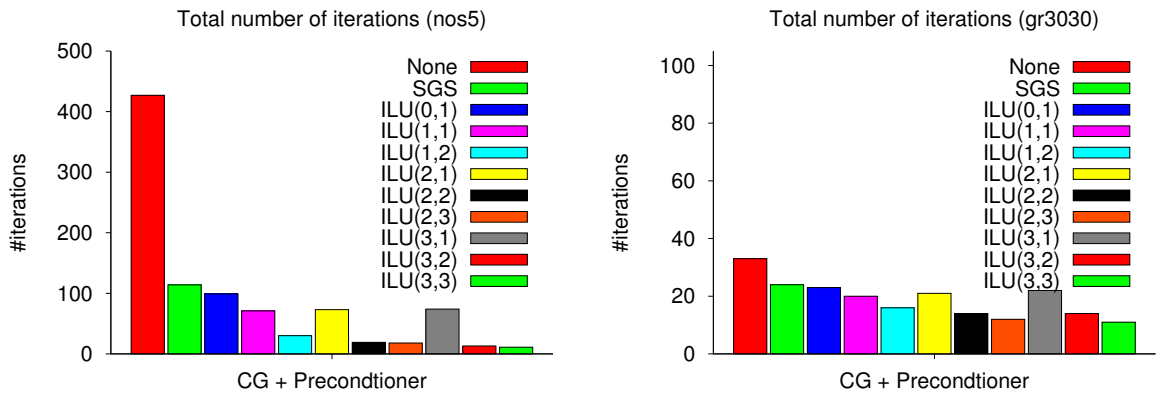

Figure 5.10: Number of CG iterations without preconditioner and with multi-colored SGS and power(q)-pattern enhanced multi-colored ILU(p,q) preconditioners for $p = 0, 1, 2, 3$ with and without drop-off for the `nos5` matrix (left) and the `gr3030` matrix (right)

When level-scheduling is applied to the factorized matrices L_p and U_p of the ILU(p) decomposition of the `nos5` matrix, we obtain 39 levels for $p = 0$, 136 levels for $p = 1$, 312 levels for $p = 2$ and 403 levels for $p = 3$. Consequently, each block contains a very low number of elements and in many cases only one element. The necessary degree of parallelism is not given in this scenario. For the `nos5` matrix, the patterns obtained by applying level-scheduling re-numbering to the ILU(p)-factorized matrices L_p and U_p (more specific $L_p + U_p$ in a single matrix structure) are shown in Figure 5.13. When the same level-scheduling re-ordering π_p corresponding to ILU(p) is applied to the original system matrix A, the patterns depicted in Figure 5.14 are observed. After re-numbering, this new structure of A is used within the parallel matrix-vector operations in the parallel CG solver. These figures show the way the locality is affected by the re-ordering of the nodes.

Now we apply level-scheduling re-ordering to the ILU(p) preconditioner for the `gr3030` matrix and obtain 87 levels for $p = 0$, 116 levels for $p = 1$, 145 levels for $p = 2$, and 174 levels for $p = 3$. The structure of the factorized matrices L_p and U_p of `gr3030` with the level-scheduling re-ordering π_p applied is shown in Figure 5.15. In this scenario, again the necessary degree of parallelism cannot be obtained by level-scheduling. When the level-scheduling re-ordering π_p corresponding to ILU(p) is applied to the original matrix A, the patterns depicted in Figure 5.16 are observed.

We can obtain a higher degree of parallelism using the power(q)-pattern method based on the $|A|^q$ matrix. For the `nos5` matrix and $q = 1, 2, 3, 4$ we observe 9, 33, 84 and 157 colors, respectively. The corresponding multi-coloring permutation is denoted by π_q. The matrix patterns $\pi_q|A|^q\pi_q^{-1}$ (not shown here) are an upper bound for the level-q fill-ins for the ILU(q, $q+1$) decomposition for $q = 0, 1, 2, 3$. The sparsity patterns of the permuted linear systems $\pi_q A\pi_q^{-1}$ are shown in Figure 5.17. These matrices are the starting points for the incomplete factorizations. In this setting, $\pi_1 A\pi_1^{-1}$ is a superset for the ILU(0,1) decomposition, see upper left figure in Figure 5.18.

For the power(q)-pattern enhanced multi-colored ILU(p,q) preconditioner, Figure 5.18 details the structure of the factorized matrices $L_{p,q}$ and $U_{p,q}$ with $A = L_{p,q}U_{p,q} + R_{p,q}$ for the `nos5` matrix. The two upper sub-figures show the factorized multi-colored ILU(p) decomposition with $p = 0$ and $p = 1$ with permutation based only on the original matrix A. The resulting matrix in the first figure is equal to the original ILU(0,1) decomposition. The matrices with more diagonal entries (i.e. matrices which require drop-off technique) are shown in the second figure. The first figure in the second row presents ILU(1,2), the factorization of first order without drop-offs. The second figure in the second row shows a drop-off strategy for $p = 2$ and $q = 1$ using only 9 colors. Results for the ILU(2,q) factorization are shown in the last row of Figure 5.18 for $q = 2$ and $q = 3$

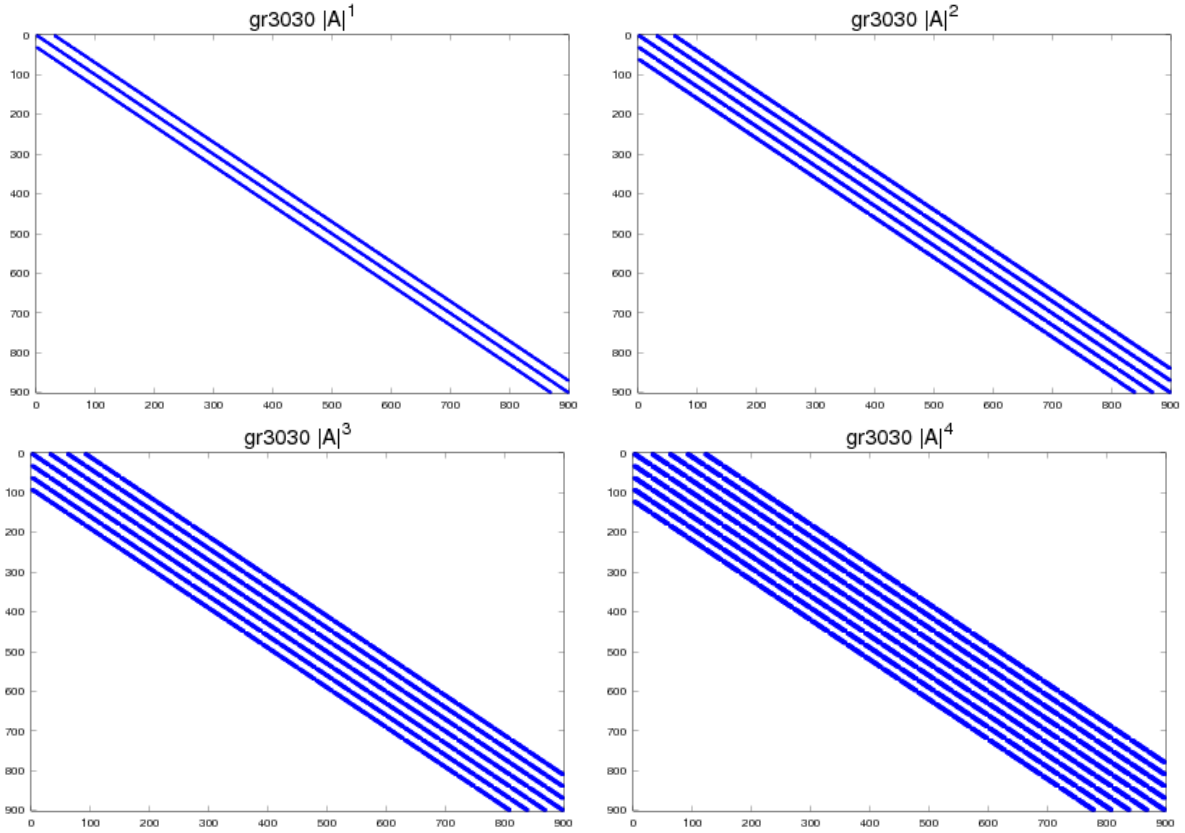

Figure 5.11: Sparsity patterns of the power $|A|^q$ of `gr3030` for $q = 1, 2, 3, 4$

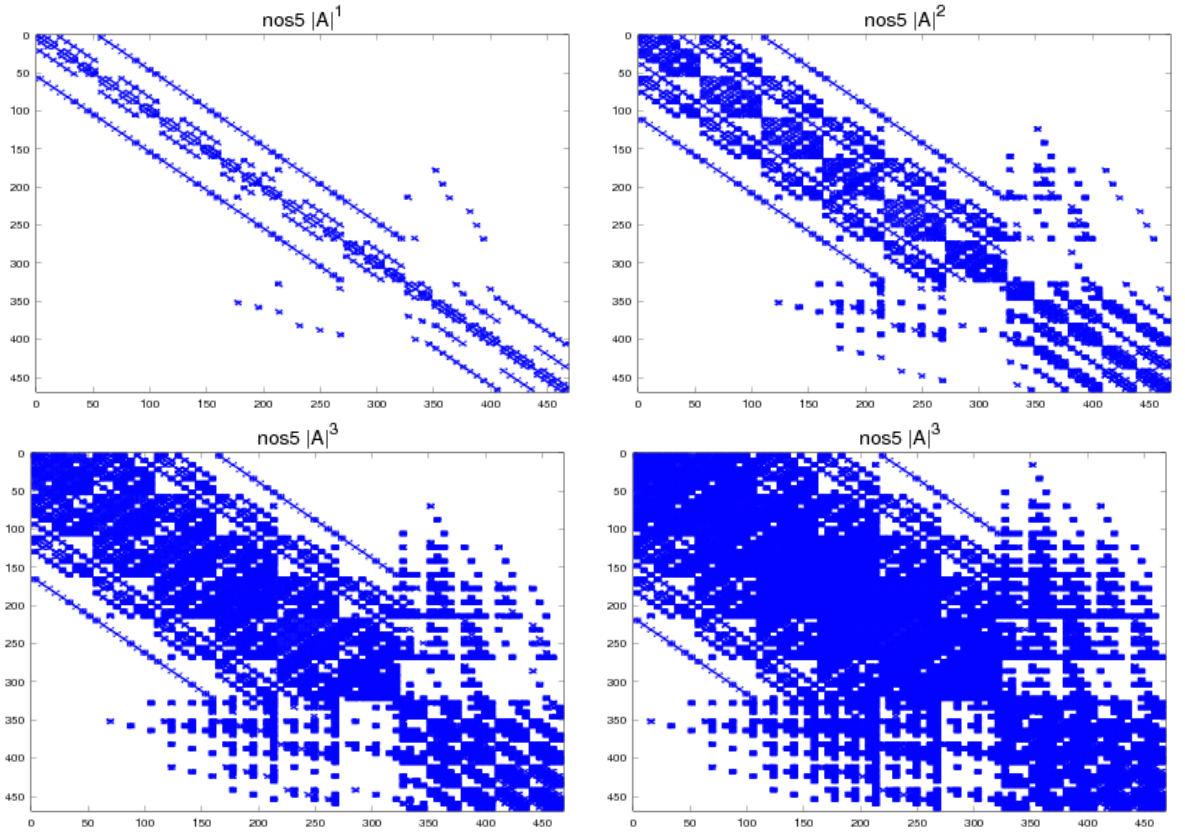

Figure 5.12: Sparsity patterns of the power $|A|^q$ of `nos5` for $q = 1, 2, 3, 4$

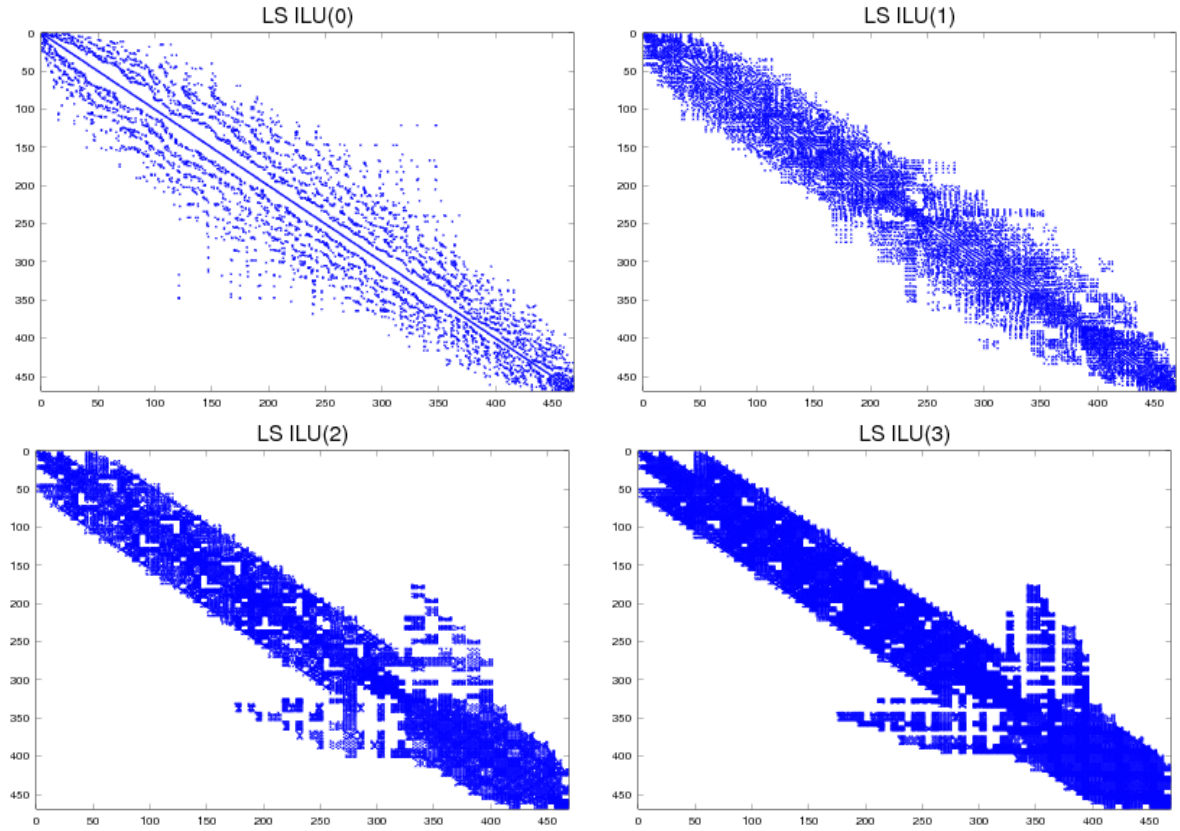

Figure 5.13: `nos5`: Level-scheduling re-ordering π_p applied to the factorized matrices L_p and U_p (combined in a single matrix structure) given by the ILU(p) decomposition with level-p fill-ins

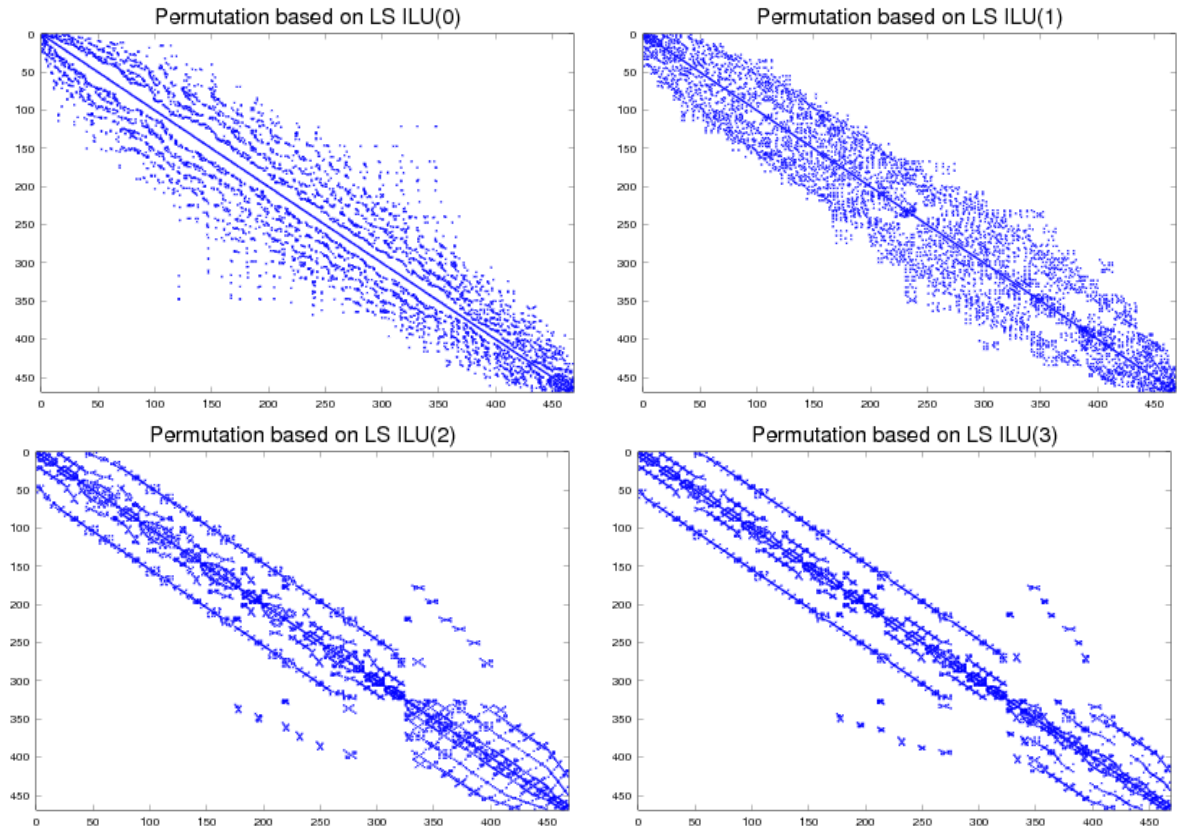

Figure 5.14: `nos5`: Level-scheduling re-ordering π_p corresponding to ILU(p) applied to the original matrix A

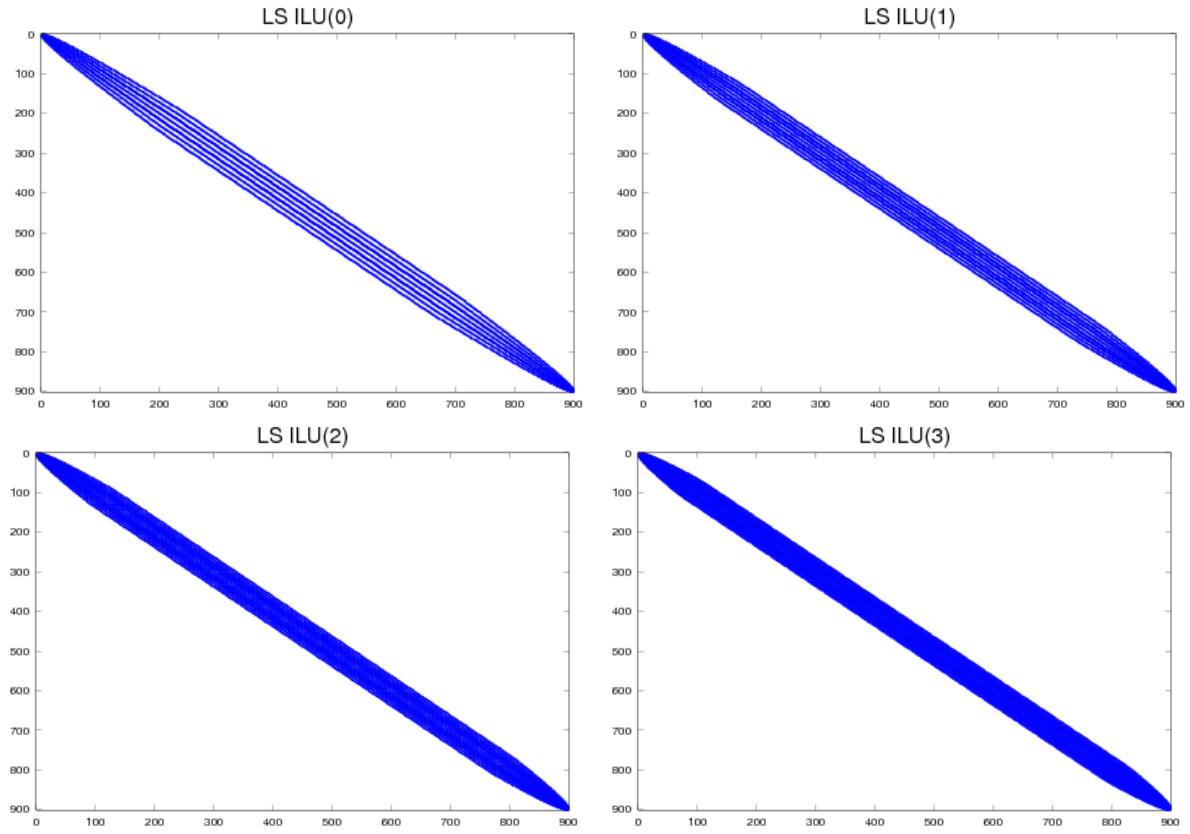

Figure 5.15: `gr3030`: Level-scheduling re-ordering π_p applied to the factorized matrices L_p and U_p (combined in a single matrix structure) given by the ILU(p) decomposition with level-p fill-ins

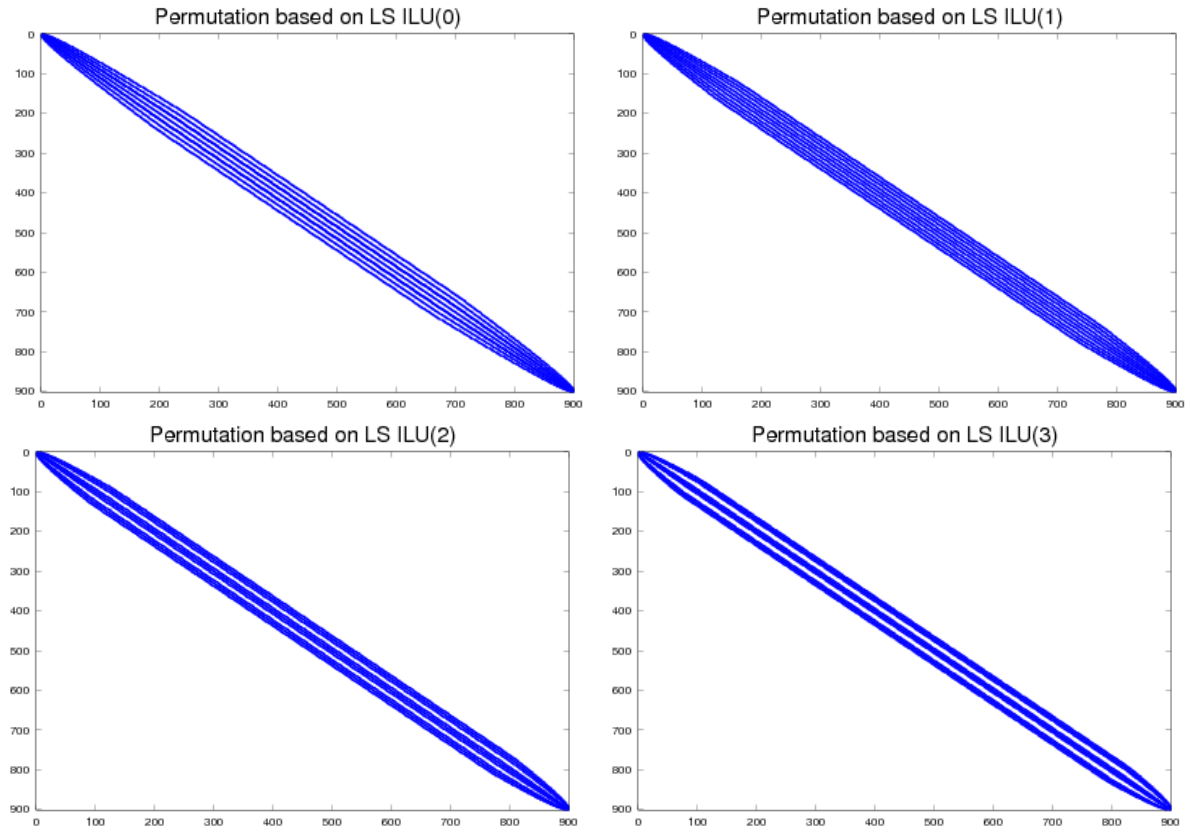

Figure 5.16: `gr3030`: Level-scheduling re-ordering π_p corresponding to ILU(p) applied to the original matrix A

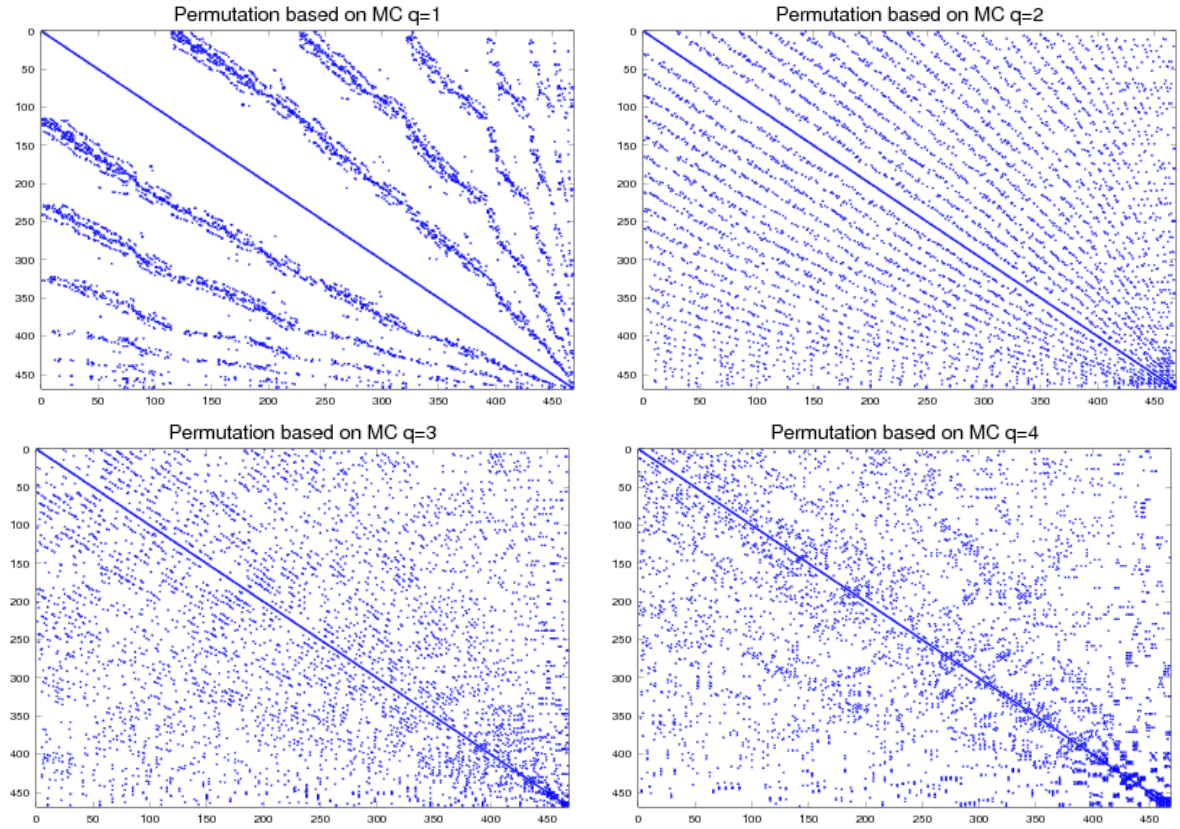

Figure 5.17: `nos5`: Sparsity patterns of the permuted linear system $\pi_q A \pi_q^{-1}$ with multi-coloring permutation π_q obtained from the analysis of $|A|^q$ for $q = 1, 2, 3, 4$ with 9, 33, 84, and 157 colors

resulting in 84 and 157 colors, where the latter case is without extra fill-ins into the diagonal blocks. Decomposition patterns for ILU(3,q) for the `nos5` matrix are depicted in Figure 5.19 for $q = 1, 2, 3, 4$. In this example, by applying the multi-coloring decomposition for $q = 1, 2, 3, 4$ we obtain 9, 33, 84 and 157 blocks, respectively.

An upper bound for the sparsity pattern for the ILU decomposition is derived by the power(q)-pattern method based on the structure of $|A|^q$. The sparsity pattern of the multi-coloring permuted system matrix, i.e. $\pi_q A \pi_q^{-1}$, for the `gr3030` matrix is depicted in Figure 5.20. Here again, π_q is obtained from a multi-color analysis of $|A|^q$. We find 4, 9, 16 and 25 colors for $q = 1, 2, 3, 4$.

For the ILU(p,q) method, Figure 5.21 details the structure of the factorized matrices for the `gr3030` matrix. The upper sub-figures show the multi-colored factorized decomposition for $p = 0$ and $p = 1$ with $q = 1$ and 4 colors. The left figure in the second row shows the ILU(1,2) factorization, the right figure in the middle row represents the drop-off strategy for ILU(2,1) with only 4 colors. The figures in the lower row present ILU(2,2) with drop-off and ILU(2,3) without drop-off. Finally, factorization matrices for the ILU(3,q) for $q = 1, 2, 3, 4$ are presented in Figure 5.22. For the `gr3030` matrix, we obtain 4, 9, 16 and 25 blocks after the multi-colored decomposition applied with $q = 1, 2, 3, 4$.

Note that in the figures representing the sparsity patterns with drop-off techniques for $q \leq p$, we present the full patterns produced after the factorization, i.e. we do not enforce deletion of the elements after the factorization phase.

5.3.4 Approximate Inverse Preconditioners

Here, we present the impact of the FSAI preconditioner on the three test matrix problems. Table 5.5 shows the number of iterations needed for solving the test systems `ecology2`, `s3dkq4m2` and `g3_circuit` with the preconditioned CG method. Furthermore, in this table we give the

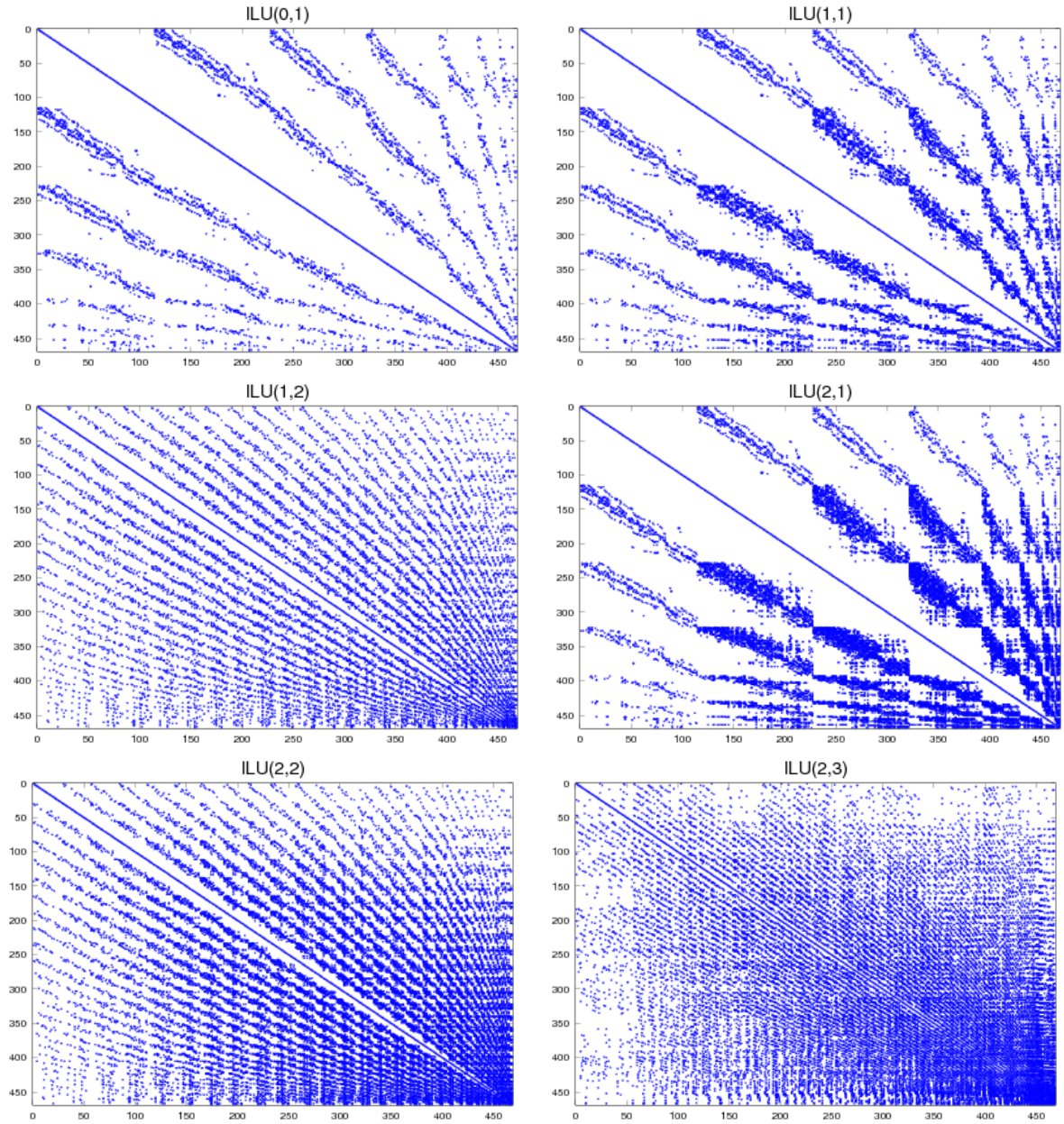

Figure 5.18: `nos5`: Sparsity patterns for the power(q)-pattern enhanced multi-colored ILU(p,q) decomposition with and without drop-off; first row: ILU(0,1) and ILU(1,1); second row: ILU(1,2) and ILU(2,1); last row: ILU(2,2) and ILU(2,3)

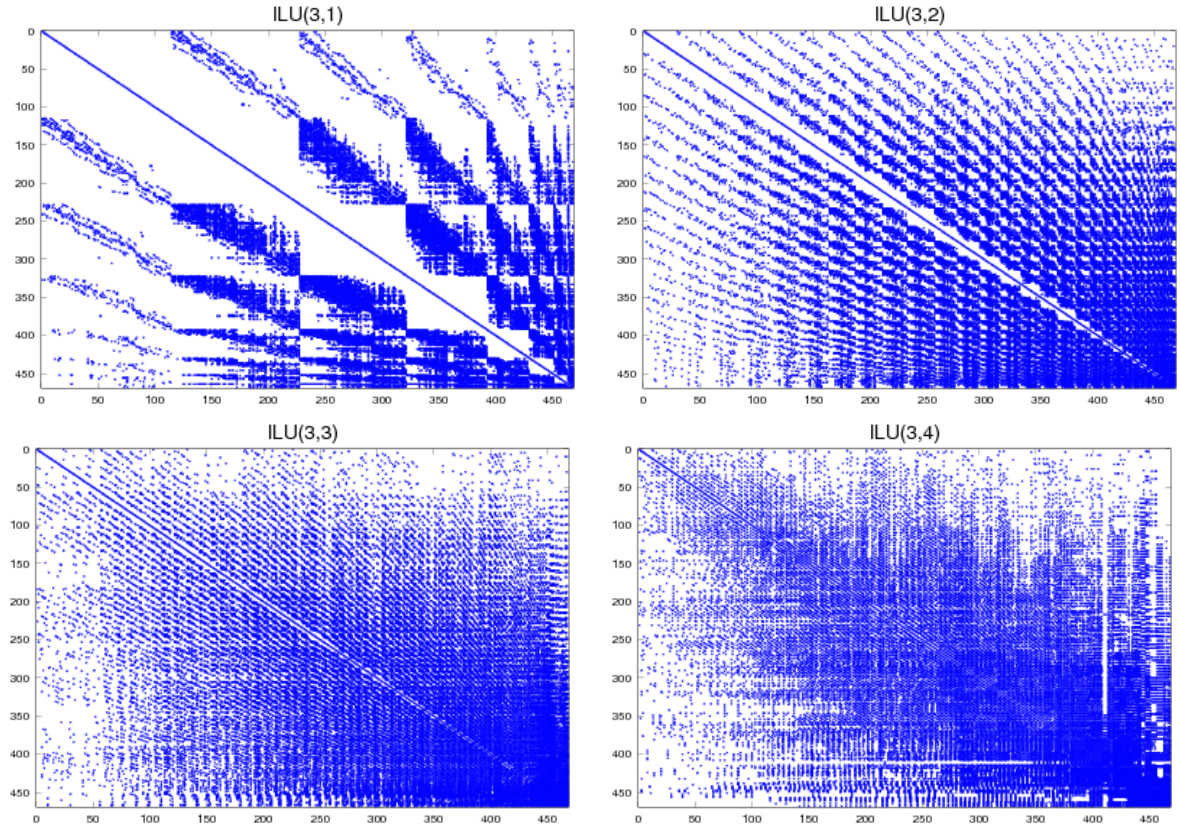

Figure 5.19: `nos5`: Sparsity patterns for the power(q)-pattern enhanced multi-colored ILU with and without drop-off; upper row ILU(3,1) and ILU(3,2); lower row: ILU(3,3) and ILU(3,4)

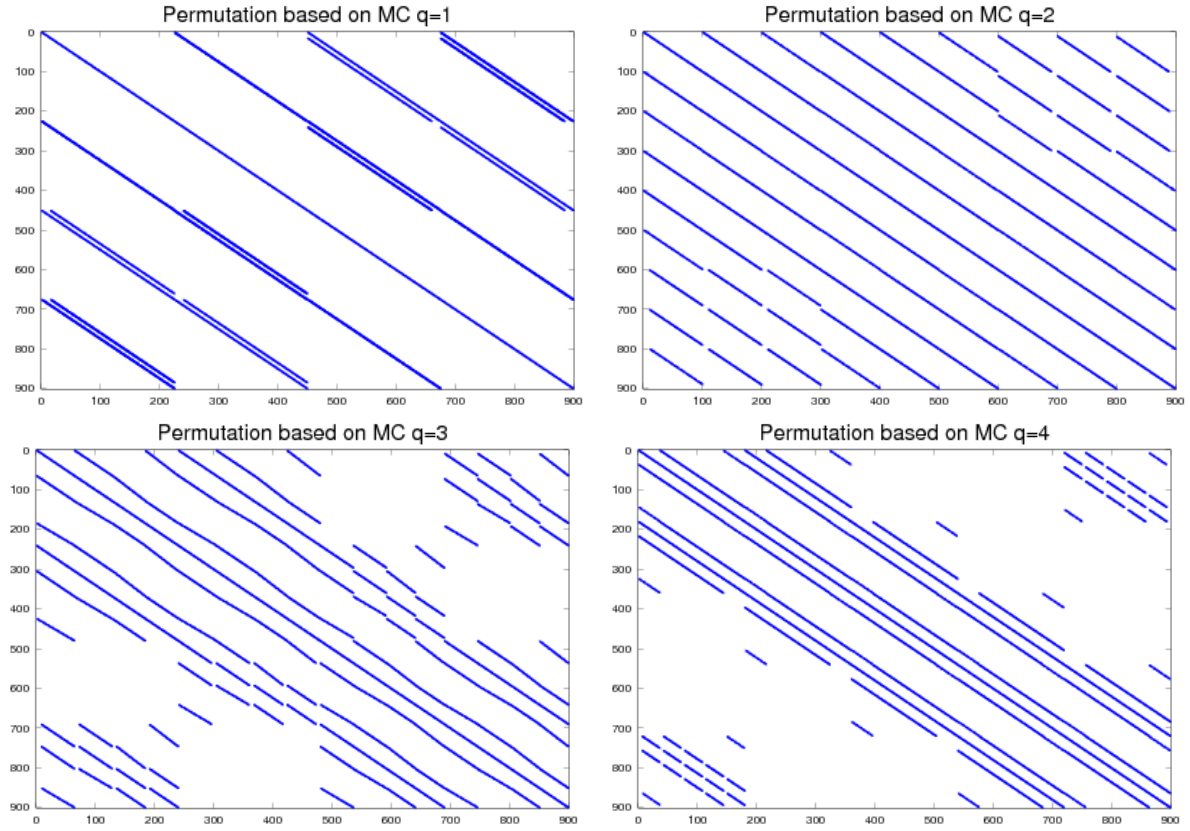

Figure 5.20: `gr3030`: Sparsity patterns of the permuted linear system $\pi_q A \pi_q^{-1}$ with multi-coloring permutation π_q obtained from the analysis of $|A|^q$ for $q = 1, 2, 3, 4$ with 4, 9, 16, and 25 colors

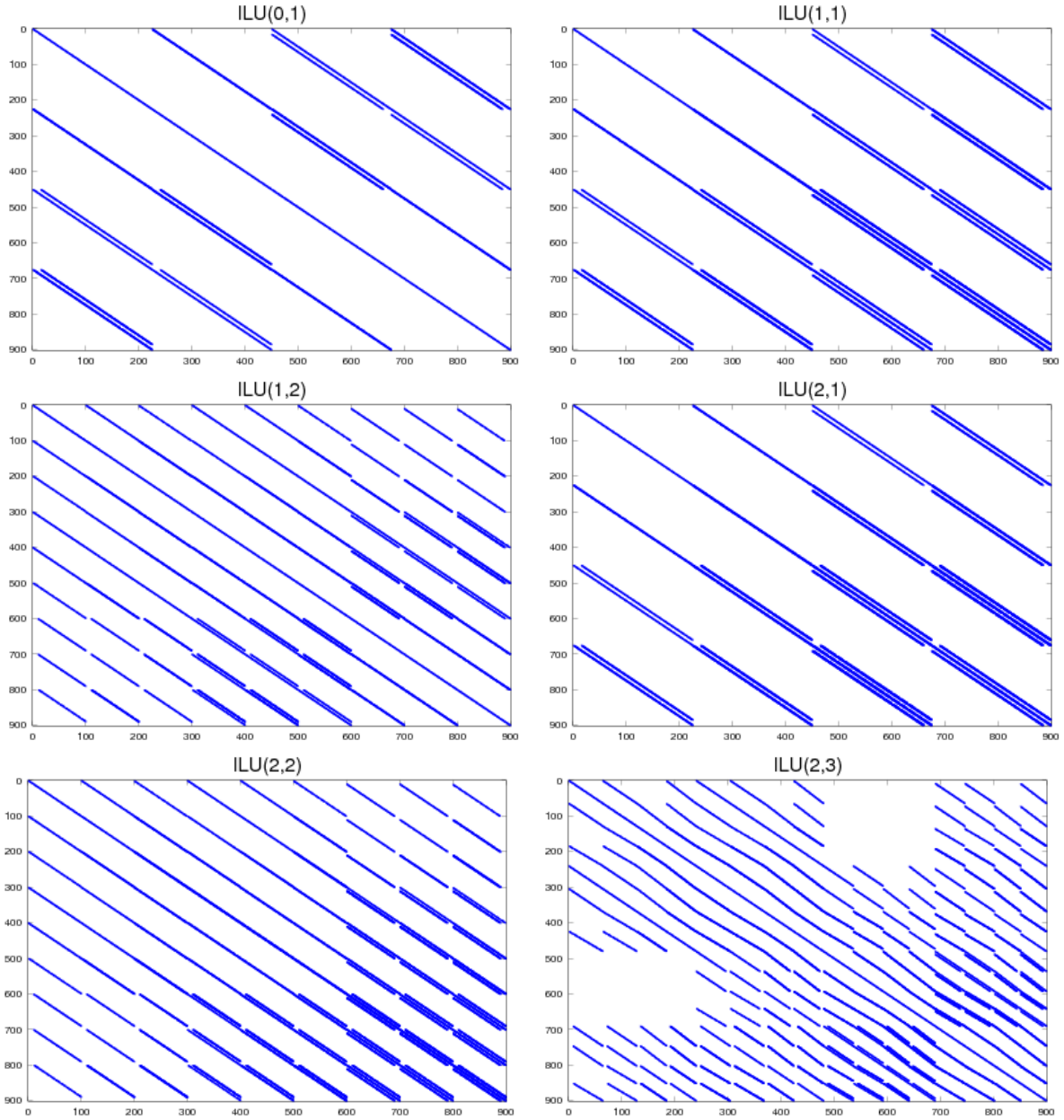

Figure 5.21: `gr3030`: Sparsity patterns for the power(q)-pattern enhanced multi-colored ILU(p,q) decomposition with and without drop-off; first row: ILU(0,1) and ILU(1,1); second row: ILU(1,2) and ILU(2,1); last row: ILU(2,2) and ILU(2,3)

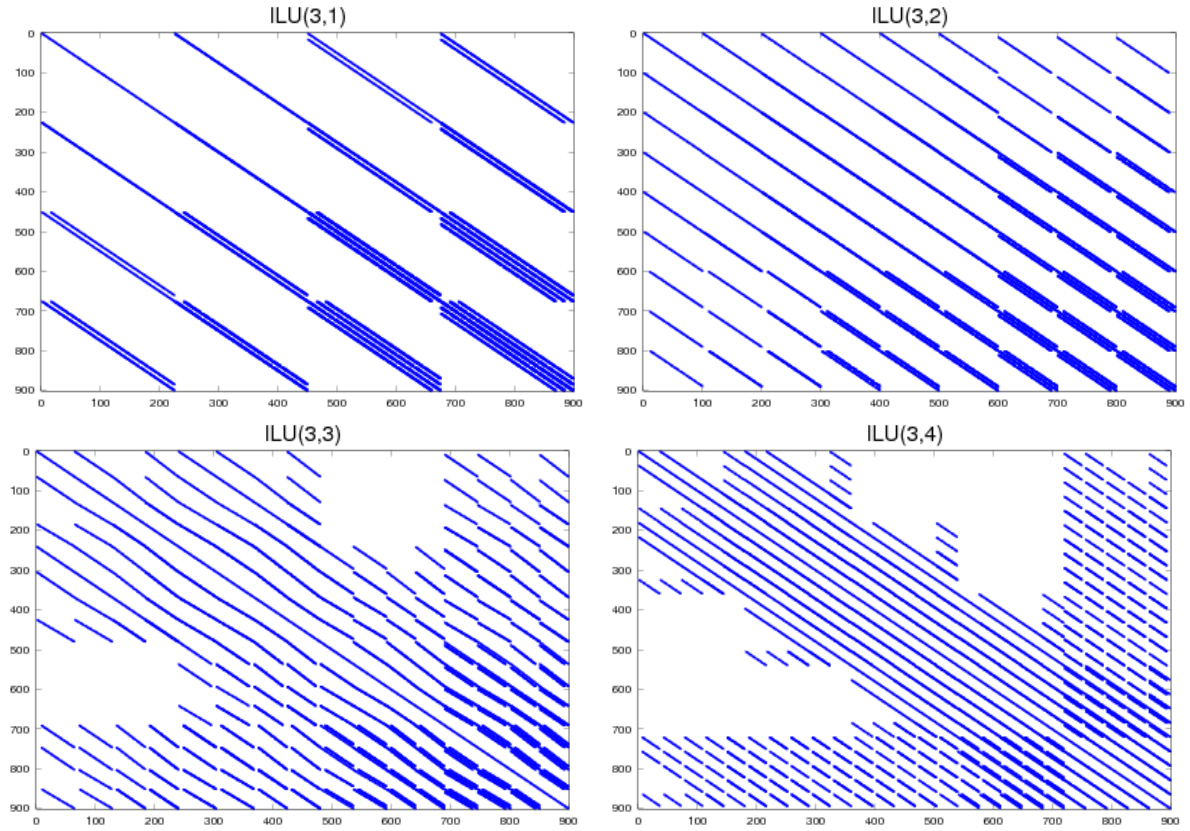

Figure 5.22: `gr3030`: Sparsity patterns for the power(q)-pattern enhanced ILU with and without drop-off; upper row ILU(3,1) and ILU(3,2); lower row ILU(3,3) and ILU(3,4)

number of non-zeroes of the preconditioned FSAI matrices $L + L^T$. To accelerate the Jacobi preconditioner, we explicitly extract the diagonal of the matrix and create a new vector out of it. Thus, we can apply the Jacobi preconditioner by performing only one component-wise vector-vector multiplication and because of that we consider the Jacobi preconditioner as an approximate inverse scheme.

		No precond	Jacobi	$FSAI_1$	$FSAI_2$	$FSAI_3$
`ecology2`	# its	5,391	5,566	2,963	2,096	1,607
	acc. fact.	1.0	0.96	1.1	2.57	3.35
	# $\mathcal{NNZ}$		999,999	5,995,990	13,979,974	25,943,958
`g3_circuit`	#its	12,760	2,726	1,309	877	648
	acc. fact.	1.0	4.6	9.74	14.5	19.96
	# $\mathcal{NNZ}$		1,585,478	9,246,304	22,852,612	48,079,052
`s3dkq4m2`	#its	535,056	36,219	3,777	2,240	1,625
	acc. fact.	1.0	14.77	141.6	238.8	329.2
	# $\mathcal{NNZ}$		90,449	4,911,340	13,373,002	25,908,490

Table 5.5: Number of iterations of the preconditioned CG solver, acceleration factors with respect to reduced iteration count, and non-zeroes entries of the approximate inverse matrices

5.3.5 Matrix-Based Multi-grid Method and Preconditioned CG

In this section, we study the properties of the parallel multi-grid smoothers based on different parallel preconditioning schemes. We investigate the smoothing properties of multi-colored ad-

ditive, power(q)-pattern and FSAI methods. To demonstrate these properties, we consider the Poisson problem (5.1) on a locally refined mesh of level 2 with 12,795 degrees of freedom, see Table 5.1 and Figure 5.23 (left).

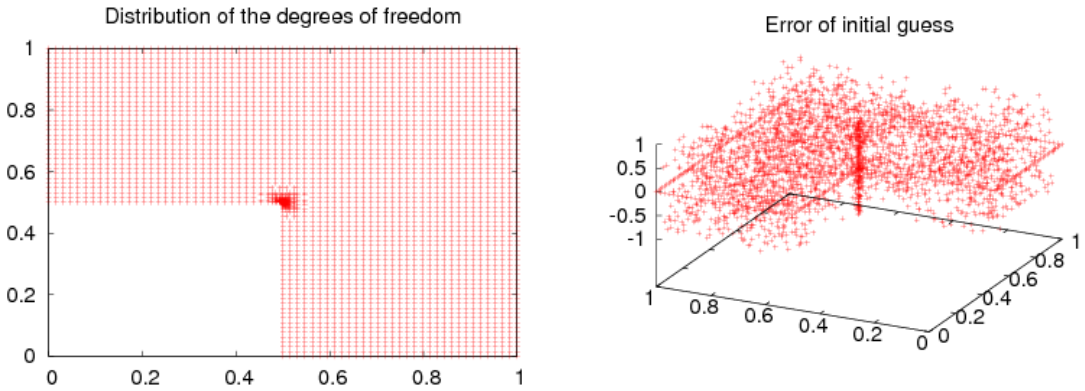

Figure 5.23: Distribution of the degrees of freedom on the L-shaped domain (left); Initial error distribution based on randomly generated initial values (right)

Figure 5.23 (right) shows the error of random initial values. In terms of reduction of the high frequency components, we see that the effect of the Jacobi smoothing, as shown in Figure 5.24, is worse than the Gauss-Seidel smoothing, presented in Figure 5.25, which itself is worse than the SGS smoother, presented in Figure 5.26.

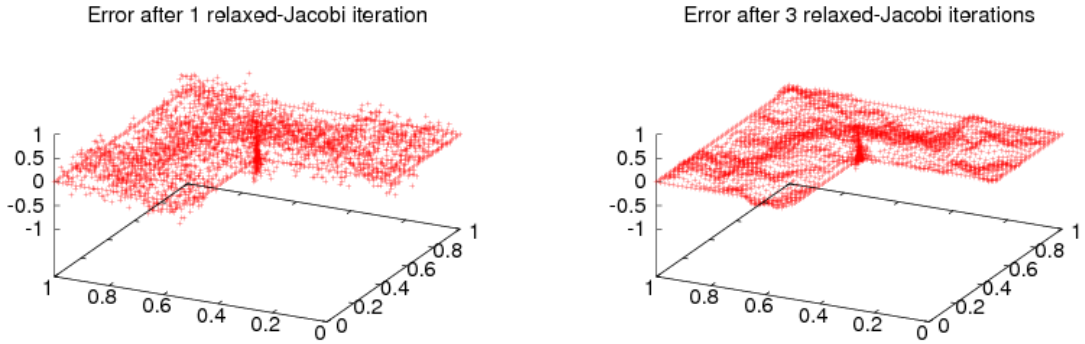

Figure 5.24: Damped error after 1 (left) and 3 (right) relaxed-Jacobi ($\omega = 0.8$) smoothing steps

More smoothing properties for ILU(0,1), ILU(1,1) and ILU(1,2) are demonstrated in Figures 5.27, 5.28 and 5.29. With the FSAI smoothers, higher order oscillations are still observed after the initial smoothing step as can be seen in Figures 5.30, 5.31 and 5.32.

With respect to the performance, we use a larger problem size which results in $3,211,425$ degrees of freedom. We perform several tests with different configurations for the pre- and post-smoothing steps. We determine the number of multi-grid cycles required to achieve a relative residual less than 10^{-6}. In Table 5.6 the iteration counts for the V-cycle based multi-grid are shown. As we can see, the standard smoothers such as Jacobi and Gauss-Seidel do not provide the best performance. Better results are obtained with the power(q)-pattern method and with the approximate inverse based on the FSAI algorithm. Note that the iteration counts in this table do not reflect the total amount of work. From a theoretical point of view, a Gauss-Seidel smoothing step is half as costly as a SGS step and ILU(0,1) is cheaper than ILU(1,1) which is cheaper than ILU(1,2). For the multi-grid solver, $\nu_1 + \nu_2$ is the number of smoothing steps on

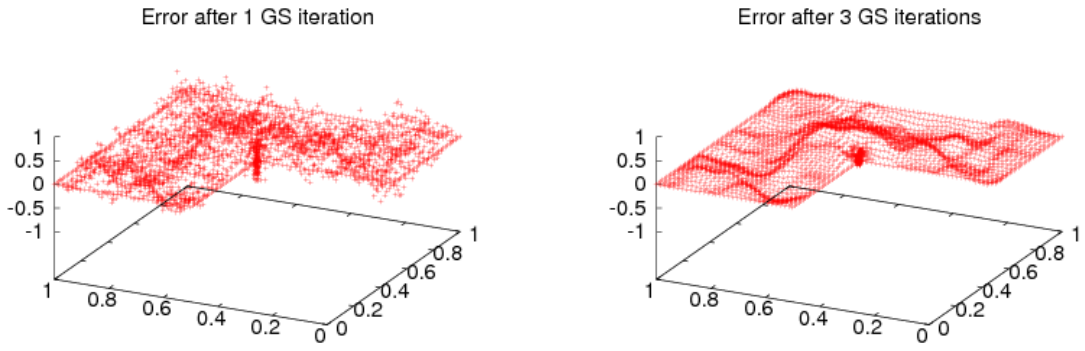

Figure 5.25: Damped error after 1 (left) and 3 (right) multi-colored Gauss-Seidel smoothing steps

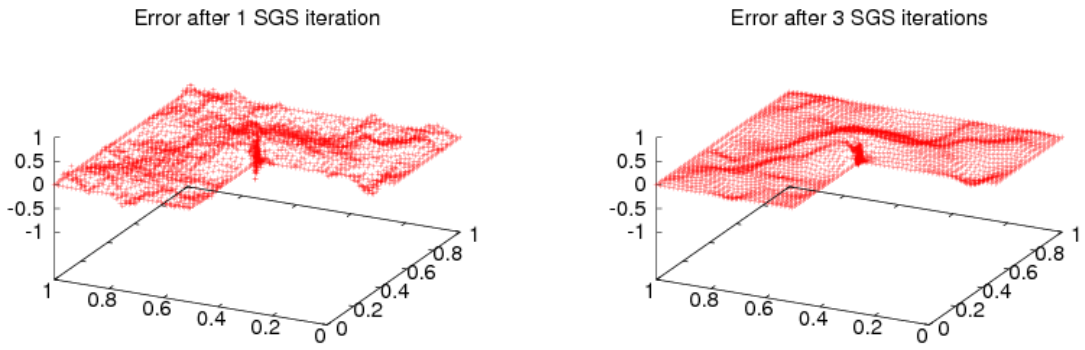

Figure 5.26: Damped error after 1 (left) and 3 (right) multi-colored SGS smoothing steps

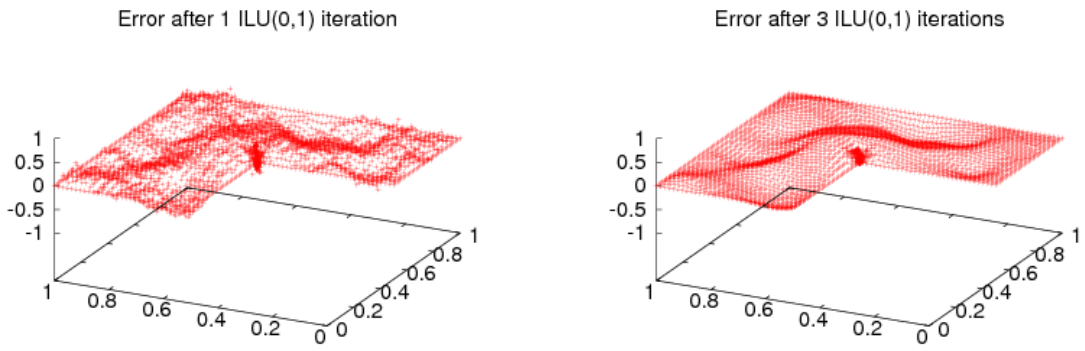

Figure 5.27: Damped error after 1 (left) and 3 (right) power(q)-pattern ILU(0,1) smoothing steps

each level and should be kept low in order to reduce the total amount of work (and hence to reduce the solver time), see Section 2.4.3.

Furthermore, in Table 5.7 we present the number of iterations of the CG solver for various preconditioners.

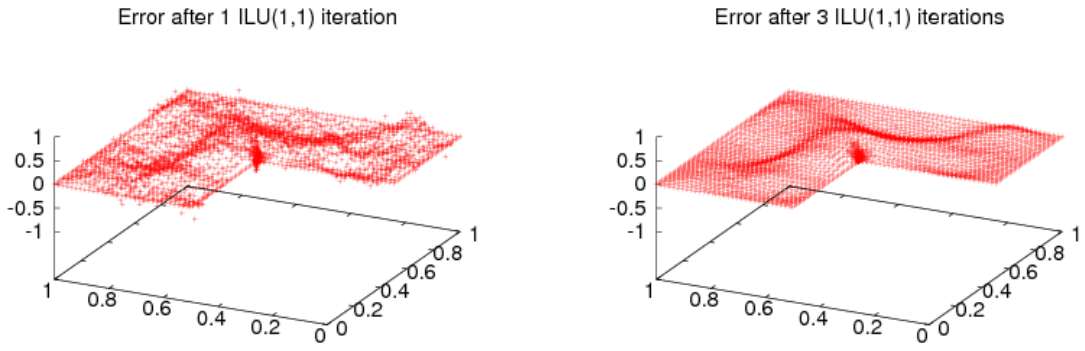

Figure 5.28: Damped error after 1 (left) and 3 (right) power(q)-pattern ILU(1,1) smoothing steps

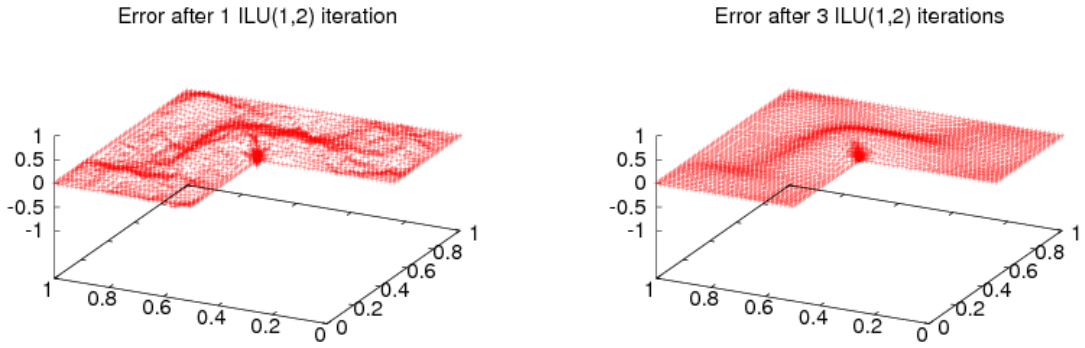

Figure 5.29: Damped error after 1 (left) and 3 (right) power(q)-pattern ILU(1,2) smoothing steps

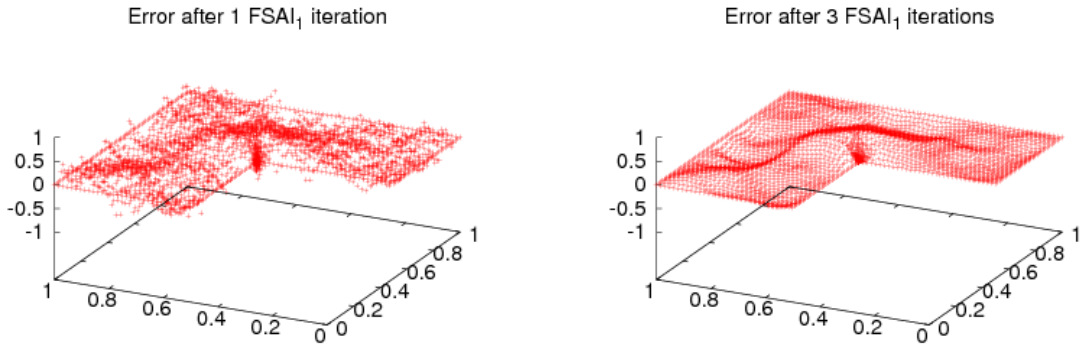

Figure 5.30: Damped error after 1 (left) and 3 (right) FSAI$_1$ smoothing steps

5.3.6 Parallelism of Level-Scheduling and Power(q)-pattern Method

As described in Section 3.6.10, the degree of parallelism of the level-scheduling method is obtained from the structure of the matrix after the factorization process. In contrast, the power(q)-pattern method is based on the multi-coloring decomposition of the matrix $|A|^q$ which is given by the topological connection of the sparsity structure. Thus, using the power(q)-pattern method for a fixed problem (i.e. a problem with given differential operator, discretization scheme, geometry and boundary condition) we obtain a certain degree of parallelism, which is independent of

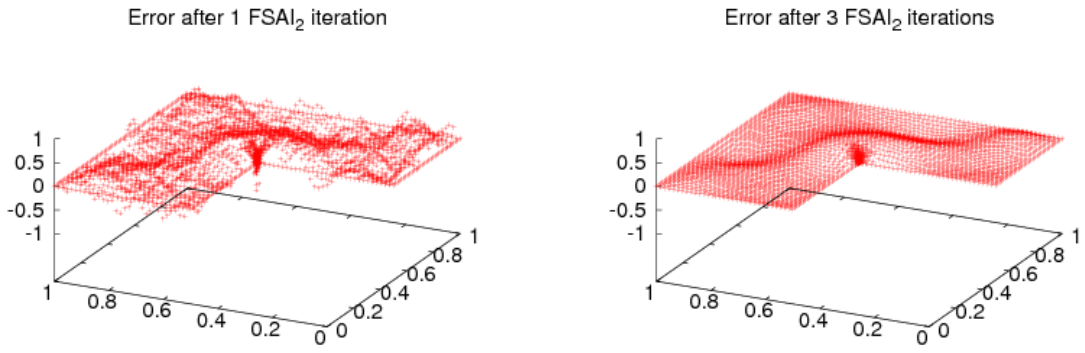

Figure 5.31: Damped error after 1 (left) and 3 (right) $FSAI_2$ smoothing steps

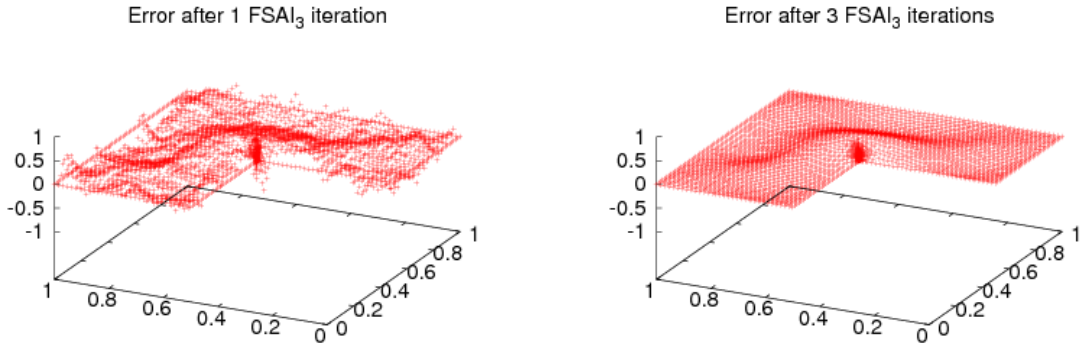

Figure 5.32: Damped error after 1 (left) and 3 (right) $FSAI_3$ smoothing steps

the number of unknowns (i.e. of the problem size). However, the degree of parallelism of the level-scheduling method decreases when we increase the problem size.

To demonstrate these aspects, we consider the 2D Poisson equation (5.1) on a square domain discretized with Q1 finite elements. The stiffness matrix is based on a lexicographical ordering and as a preconditioner we use ILU factorization without fill-ins. Furthermore, we discretize the problem on 200-by-200; 500-by-500 and $1,000$-by-$1,000$ grids which results in stiffness matrices with sizes $40,000$; $250,000$ and $1,000,000$. Thus, with the power(q)-pattern method we obtain only 2 colors in all of the problems, while with the level-scheduling method we obtain 399, 999 and $1,999$ levels for the different problem sizes.

In general, it is not possible to compare the quality of the ILU factorization based on the level-scheduling and the power(q)-pattern method. The resulting operators in these cases are different which leads to different iteration numbers for the solution procedure. In Appendix B, we present several test matrices in order to make a comparison. We observe that the power(q)-pattern method requires more iterations than the level-scheduling algorithm based on the original ordering and factorization. However, combined with a much higher degree of parallelism, the solvers perform faster on all of the parallel system when using the power(q)-pattern method, see Appendix D.

5.4 Performance Analysis

Up to now, we have considered only the efficiency of the parallel preconditioned linear solvers in terms of number of iterations. In this section, we study the performance behavior of the

(ν_1, ν_2)	(0,1)	(0,2)	(0,3)	(0,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(2,0)	(2,1)	(2,2)
r-Jacobi	64	32	21	16	71	34	22	17	13	38	24	18
GS	30	14	10	8	35	16	10	8	7	20	12	9
SGS	23	13	9	8	28	14	10	8	7	17	11	9
ILU(0,1)	18	11	8	6	25	11	8	7	6	15	8	7
ILU(1,1)	18	10	7	6	23	11	8	7	6	14	8	7
ILU(1,2)	10	6	5	5	12	7	5	5	5	9	6	5
$FSAI_1$	17	9	7	6	22	10	7	6	6	14	8	7
$FSAI_2$	14	7	6	5	15	8	6	5	5	10	6	6
$FSAI_3$	9	6	5	5	12	6	5	5	5	8	6	5

(ν_1, ν_2)	(2,3)	(2,4)	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)
r-Jacobi	14	12	27	19	15	12	11	22	17	13	11	10
GS	7	7	15	10	8	7	6	12	9	8	7	7
SGS	7	7	13	9	8	7	6	11	9	7	7	6
ILU(0,1)	6	6	11	7	6	6	6	9	7	6	6	5
ILU(1,1)	6	5	10	7	6	6	5	9	6	6	5	5
ILU(1,2)	5	5	8	6	5	5	5	7	6	5	5	5
$FSAI_1$	6	6	11	8	6	6	6	10	7	6	6	6
$FSAI_2$	5	5	8	6	6	5	5	8	6	5	5	5
$FSAI_3$	5	5	8	6	5	5	5	7	5	5	5	4

Table 5.6: Number of multi-grid V-cycles for different smoothers; ν_1 and ν_2 are the numbers of pre- and post-smoothing steps, $\omega = 0.8$ is the relaxation parameter for the Jacobi scheme

	No precond	Jacobi	SGS	ILU(0,1)	ILU(1,1)	ILU(1,2)	$FSAI_1$	$FSAI_2$
# iterations	5,650	4,167	2,323	2,451	2,066	1,387	2,198	1,493

Table 5.7: Number of iterations for solving the Poisson problem on the L-shaped domain with the preconditioned CG solver

preconditioned solvers on multi-core CPU and GPU devices in terms of execution time and speed up factors.

5.4.1 Performance Model

As shown in Section 2.6, the effective performance upper bound in terms of Flop/sec is given by

$$P_{\text{eff}} \leq \min \left\{ P, \frac{fB}{Sw} \right\},$$

where S is 4 or 8 bytes for single precision or double precision data respectively, B (in Byte/sec) is the maximal bandwidth between memory and cores, P is the peak performance of the system in terms of Flop/sec and f is the number of floating point operations for executing the specific routine, see Table 2.1.

It is interesting to note that even if the computing device does not provide fully pipelined single/double precision support or if its peak performance is not very high, for most hardware setups the ratio fB/Sw for the basic vector-vector and sparse matrix-vector routines is far below the peak performance P of the machine. Thus, any algorithm based on these routines is more dependent of the bandwidth performance than of the actually floating-point performance.

(ν_1, ν_2)	(0,1)	(0,2)	(0,3)	(0,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(2,0)	(2,1)	(2,2)
r-Jacobi	56	28	19	15	59	29	19	15	12	32	20	15
GS	25	13	9	7	28	13	9	7	6	15	10	8
SGS	23	12	9	7	25	13	9	7	6	14	9	7
ILU(0,1)	19	11	8	6	21	11	8	6	5	12	8	6
ILU(1,1)	18	10	8	6	20	11	8	6	6	12	8	7
ILU(1,2)	10	6	5	4	11	7	5	4	4	7	5	4
$FSAI_1$	18	9	6	5	19	10	7	5	5	10	7	6
$FSAI_2$	14	7	5	4	15	8	5	4	4	8	6	5
$FSAI_3$	9	5	4	4	10	6	4	4	4	6	5	4

(ν_1, ν_2)	(2,3)	(2,4)	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)
r-Jacobi	12	10	22	16	13	11	9	17	13	11	9	8
GS	6	6	10	8	7	6	5	9	7	6	5	5
SGS	6	6	10	8	7	6	5	8	7	6	6	5
ILU(0,1)	5	5	8	7	6	5	4	7	6	5	5	4
ILU(1,1)	6	5	9	7	6	5	5	7	6	5	5	4
ILU(1,2)	4	4	6	5	4	4	4	5	4	4	4	4
$FSAI_1$	5	5	8	6	5	5	4	7	5	5	5	4
$FSAI_2$	4	4	6	5	4	4	4	6	4	4	4	4
$FSAI_3$	4	4	5	4	4	4	3	5	4	4	4	3

Table 5.8: Number of multi-grid W-cycles for different smoothers; ν_1 and ν_2 are the numbers of pre- and post-smoothing steps, $\omega = 0.8$ is the relaxation parameter for the Jacobi scheme

5.4.2 Hardware Configuration

Our performance benchmarks are based on a CPU-GPU configuration with a dual-socket Intel Xeon (E5450) quad-core platform equipped with an NVIDIA Tesla S1070 system which provides two GPUs. On this system we run 64-bit Ubuntu Linux 10.04 with kernel 2.6.32-33. We use gcc 4.3.4 compiler and NVIDIA CUDA compiler version 2.3. The memory capacities of the CPU and the GPU platforms are 16 GBytes and 2×4 GBytes, respectively. The two CPUs are connected to the memory via UMA memory system. Each core has 32 KBytes of L1 cache and two cores share 6 MBytes of cache. This leads to 12 MBytes per CPU and thus 24 MBytes in total.

With a single core of the CPU we can utilize 2.62 GByte/s of the total bandwidth. Using all the eight cores with the UMA architecture, we can obtain 6.14 GByte/s. Thus, comparing the sequential version with the OpenMP version of the linear solver on the same CPU configuration, we expect speed up factors around 2.34 (=6.14/2.62). For small size problems, we observe a larger speed up factor due to cache-effects when using multiple cores and aggregating their caches. However, for very large size problems, all the data has to be streamed in and out of the core and thus we cannot exceed this ratio. Performance envelopes for more advanced model based on instruction pipelines, fused routines and Single Instruction Multiple Data (SIMD) can be found in [39]. Matrix-vector multiplication performance on CPU systems are considered in [132].

The bandwidth for performing the vector update routines on the GPU device is about 83.3 GByte/s. However, this is a rough estimation – for small vectors and matrices, the reduction based routines (e.g. SpMV and scalar product) perform faster on cache-memory based systems due to better handling of the temporary data and results. For different sparsity patterns, SpMV routines in CSR format typically perform around 1/4 to 1/15 of the peak bandwidth performance [5, 19].

The NVIDIA S1070 GPU has a texture memory which can be used efficiently for read-only caching. The SpMV can benefit from this cache mechanism by keeping some of the elements of the vector locally, see [19]. However, for matrices in CSR format this speed up is only about 6-12%. For small matrices we observe a performance decrease due to the extra time for binding and unbinding memory buffers to the texture memory.

The performance ratio of the GPU and the CPU bandwidth based on the stream benchmarks is 13.56(= 83.3/6.14). However, in practice this value is lower – even for large data sets the CG algorithm executed on all the eight cores of the CPU is about 2 to 5 times slower than its GPU version. This is mainly attributed to the good caching effects of the CPU – in the CG solver, after updating the residual vector or after executing the matrix-vector multiplication, most of the data is still kept in the cache and the scalar product of these vectors is computed fast, see Algorithm 1.

All experiments and benchmarks are performed with double precision – on both the CPU and GPU device.

5.4.3 Poisson Problem – Multi-Grid and CG Solver

The best run times for solving the Poisson problem on the locally refined 2D L-shaped domain with V and W-cycle multi-grid are presented in Figure 5.33. The run time of the multi-grid solver is equal to the time for traversing the grids and solving the coarsest system plus the time for performing the pre- and post-smoothing steps. Thus, the compute time is a function of the quality of the smoothers and their execution time. Details on the execution of the multi-grid solver based on different pre- and post-processing steps can be found in [59]. The best run times show that the multi-grid solver based on W-cycle is 2 to 3 times slower than the one based on V-cycle, which is in contrast to the number of iterations. This can be explained by the fact that the W-cycle multi-grid performs most of the operations on the coarser grids. As previously mentioned, GPU platforms are not suitable for operations on small amounts of data. A larger number of SpMV routines calls for small matrices causes a significant overhead of the GPU. However, this does not reflect the speed up factors on the CPU. On the other hand, the performance of the multi-grid V-cycle scales very well on the CPU and the GPU, see Figure 5.34. The best results are obtained with the ILU(1,2) and ILU(0,1) smoothers for V-cycle based multi-grid solver on the GPU .

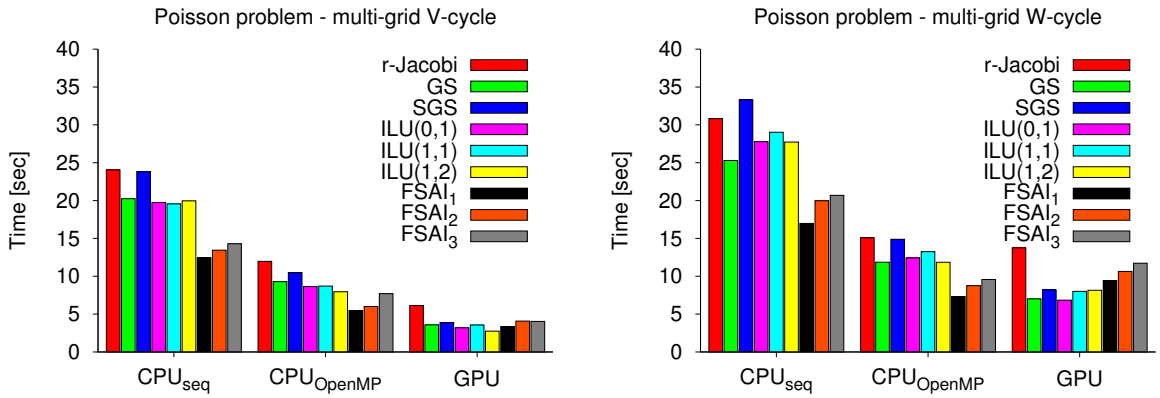

Figure 5.33: Run times for the multi-grid V-cycle (left) and W-cycle (right) for the Poisson problem on the 2D L-shaped domain

The run times for the CG are presented in Figure 5.35 (left) and the parallel speed up factors in Figure 5.35 (right). However, due to the large number of iterations, the CG is 50 to 75 times slower than the V-cycle based multi-grid solver.

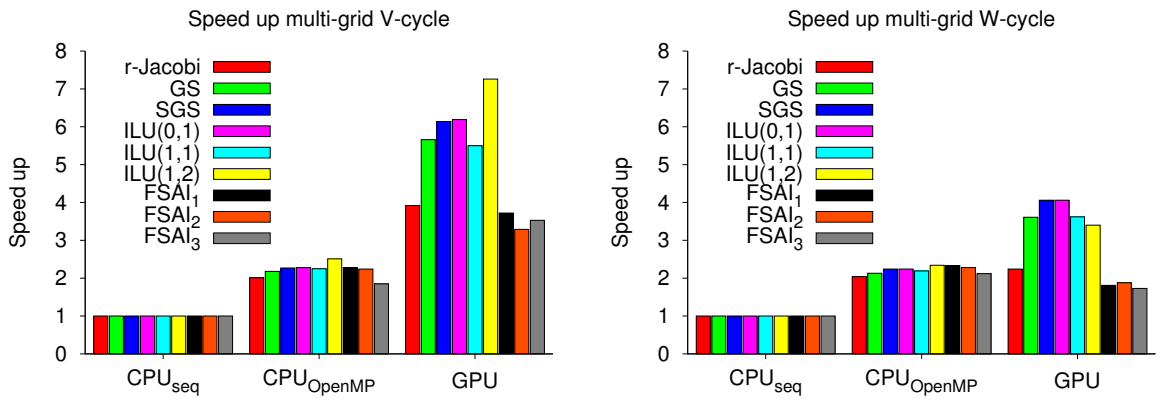

Figure 5.34: Parallel speed ups of the multi-grid V-cycle (left) and W-cycle (right) for the Poisson problem on the 2D L-shaped domain

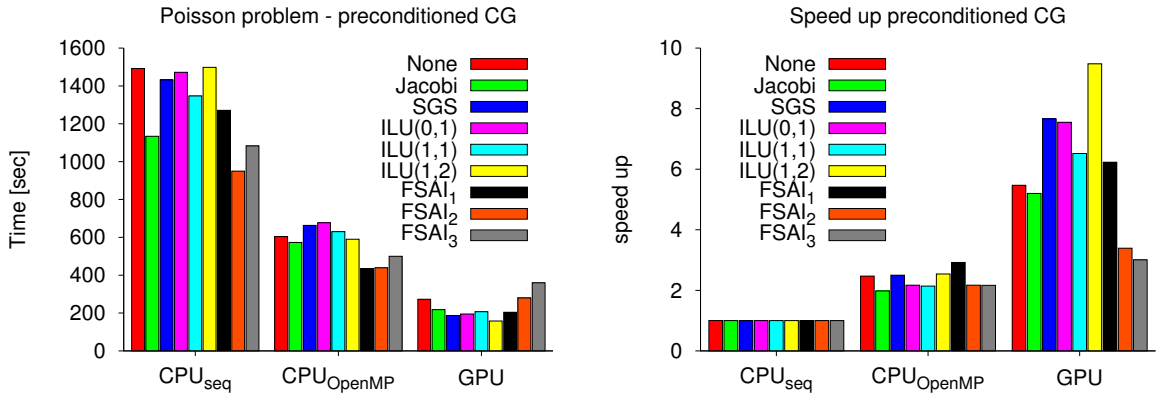

Figure 5.35: Run time performance and parallel speed up factors of the preconditioned CG solver for the Poisson problem on the 2D L-shaped domain

The stiffness matrix is not built using a lexicographical order but based on two loops – an iteration loop over the finite element cells and a loop over the degrees of freedom, for additional details see [4]. This is the default enumeration for the degrees of freedom in HiFlow^3. However, this ordering technique does not improve the performance on the stream based devices like GPUs – combined with less efficiency of this scheme, we see poor speed ups for the FSAI preconditioners based on $|A|^1$, $|A|^2$ and $|A|^3$. On the other hand, the CPU manages to cache most of the memory accesses and hence we see good speed ups with the OpenMP backend. The performance profiles for the GPU are different for the multi-coloring based preconditioners where the unknowns are clustered in groups with non-adjacent neighbors and therefore the matrix operations are better then on the CPU backend.

5.4.4 Convection-Diffusion Problem – GMRES Solver

The effect of the preconditioning for the GMRES method is presented in Figure 5.8. The figure depicts the necessary number of iterations needed to achieve a prescribed error tolerance for the GMRES method. We find out that all the four preconditioners under consideration – multi-colored Gauss-Seidel, multi-colored ILU decomposition with fill-level zero, two blocks for the block-Jacobi with block-level multi-colored Gauss-Seidel (BJ-GS), and two blocks for the block-Jacobi with block-level multi-colored ILU (BJ-ILU) – decrease the number of necessary iterations. Figure 5.36 presents the run times for the preconditioned GMRES solver in the two-dimensional (left) and the three-dimensional case (right). Each figure contains performance results for the

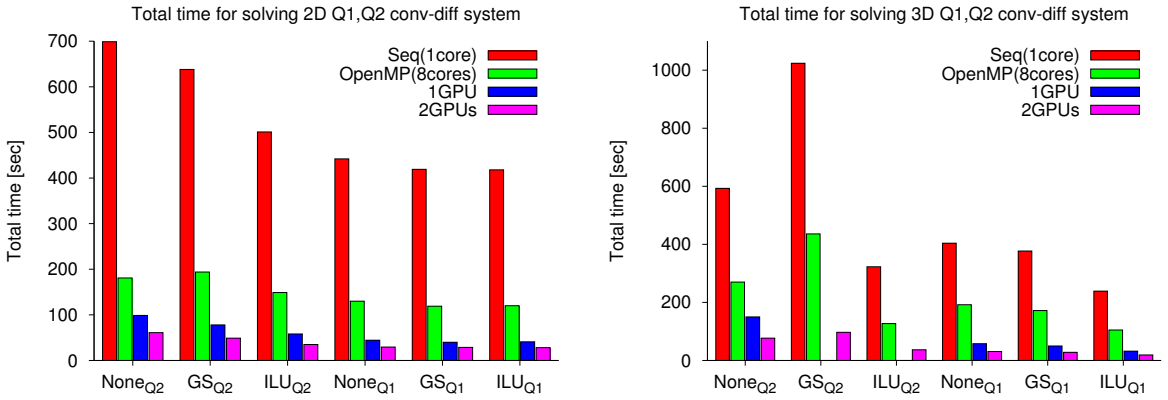

Figure 5.36: Run times for solving the 2D (left) and the 3D (right) convection-diffusion problem with Q1 and Q2 finite elements

single CPU core, eight CPU cores, single GPU and dual GPU configuration. The best results in terms of speed ups are observed for the multi-colored ILU preconditioner running simultaneously on two GPUs. Figure 5.37 depicts the speed ups due to the parallel execution on multiple cores and on the single/dual GPU(s), respectively.

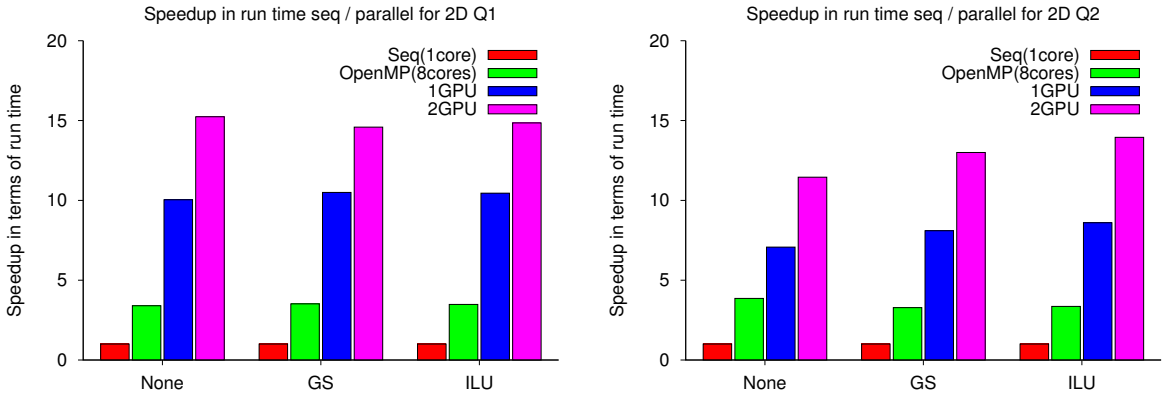

Figure 5.37: Speed up factors based on the parallel execution of the (non-)preconditioned GMRES solver on the single/eight-core CPU and on the single/dual GPU(s) platform

We use sequential routines for the single core tests and OpenMP parallel routines with explicit thread pinning on the eight-core CPU configuration. On the GPU configuration we use CUBLAS routines for the BLAS1 vector operations and scalar CSR SpMV kernels for the SpMV. For the dual GPU configuration we use two MPI processes for accessing two separate memory spaces on the GPUs. For the 2D problem, texture caching on the GPU increases the performance for both Q1 and Q2 elements. Since for the multi-colored forward and backward steps the number of SpMV grows quadratically with respect to the colors, the call overhead of SpMV kernels gets more pronounced. In the 3D case with Q1 elements (13 colors), texture caching still improves the performance but for Q2 elements (35 colors) there is no benefit of using it due to the large amount of SpMV in the forward/backward sweeps. The preconditioned system for the 3D case with Q2 elements exceeds the memory capacity of the GPU device.

5.4.5 Linear System Problems Based on Matrix Collection – CG Solver

In this section, we investigate and present the efficiency and the scalability of several fine-grained parallel preconditioners in terms of solver times, parallel speed ups, and acceleration factors due to preconditioning (for a fixed platform and implementation). In particular, we show how the power(q)-pattern enhanced multi-colored ILU(p,q) preconditioner behaves for different values of p and q, where $q = p + 1$ represents the normal scenario and $q < p + 1$ represents the drop-off technique with a reduced number of colors. In this case, the non-diagonal elements in the diagonal blocks are deleted, see [62]. We also show performance profiles for other preconditioners such as multi-colored SGS, Jacobi and FSAI.

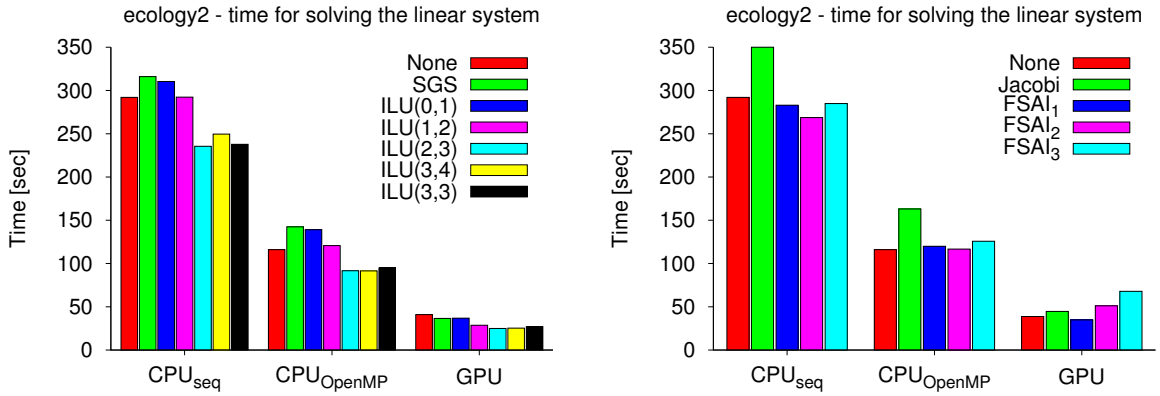

Figure 5.38: `ecology2` matrix: Performance benchmarks on the single/eight-core CPU and on the GPU. Solver time for the multi-colored SGS and the power(q)-pattern enhanced ILU(p,q) (left), Jacobi and FSAI algorithms based on $|A|^1$, $|A|^2$ and $|A|^3$ (right)

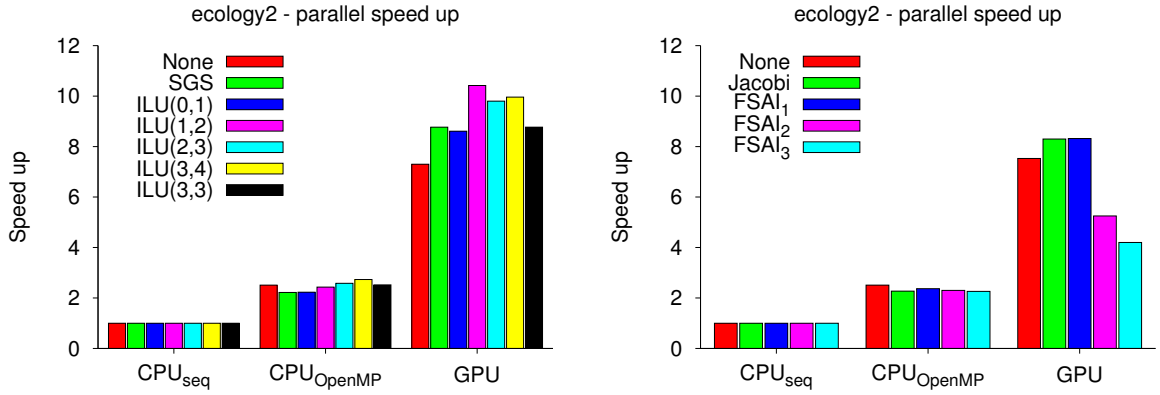

Figure 5.39: `ecology2` matrix: Parallel speed ups for various preconditioned CG solvers on the CPU OpenMP and the GPU platform

Figure 5.38 details the performance of the preconditioner for the `ecology2` problem. The performance is achieved with and without a preconditioner to reach the prescribed error tolerance. We consider implicit preconditioners such as the multi-colored SGS preconditioner with two colors, the multi-colored ILU(0,1) decomposition with two colors, and the power(q)-pattern method enhanced multi-colored ILU(p,q) method with level-p fill-ins for $p = 1, 2, 3$, see Figure 5.38 (left). The matrix exponent q (with $q = p+1$ or $q < p+1$) for determining the sparsity pattern is given in the notation ILU(p,q). For $q < p+1$, all fill-in elements within the diagonal blocks are deleted. The figure shows the corresponding solver times including the preconditioning step (triangular sweeps) but not the preprocessing step (LU factorization, multi-coloring or power(q)-pattern

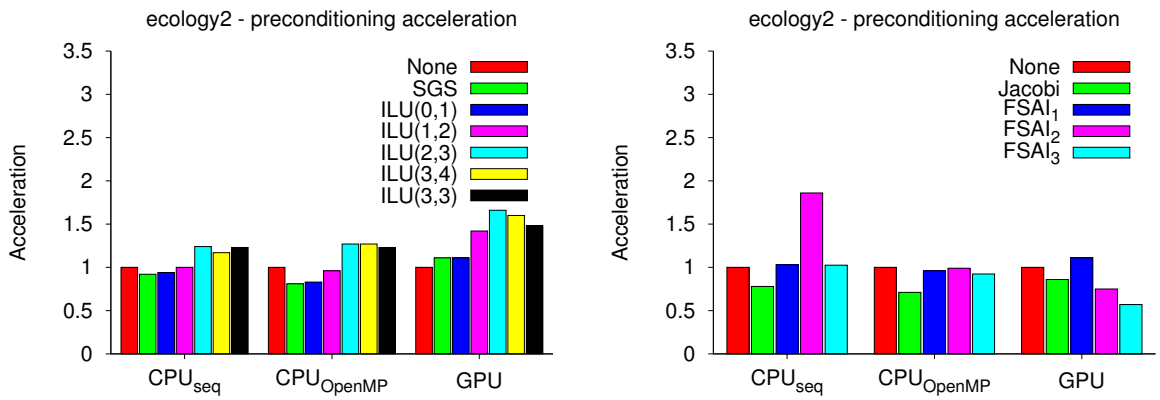

Figure 5.40: `ecology2` matrix: Acceleration factors for the preconditioning phase in the CG solver on the single/eight-core CPU and the GPU platform

determination). Furthermore, we consider explicit preconditioners such as Jacobi and the FSAI algorithm based on the sparsity patterns of $|A|^q$ (denoted with FSAI_q), see Figure 5.38 (right). The figures depict solver times for the sequential solution running on a single core, the OpenMP parallel solution running on eight-core, and the single GPU version with texture caching for the non-preconditioned solver and without texture caching for the preconditioned one.

In Figure 5.39 the associated speed ups for the `ecology2` matrix are presented. The figure shows the parallel speed ups for the eight core OpenMP parallel version and the data-parallel GPU version. The GPU version is, by a factor of 3 to 4, faster than the OpenMP parallel version on the eight CPU cores. Figure 5.40 shows the overall acceleration of the preconditioned solver over the non-preconditioned version with a fixed hardware configuration and implementation.

For this test problem, the preconditioner on the GPU accelerates the solver by a factor of up to 1.7 for the ILU(2,3). In some cases, there is almost no acceleration by preconditioning due to the additional work and minimal benefits from reduced iteration count. The drop-off technique for ILU(3,3) reduces the number of colors from 19 to 8, slightly increases the number of iterations but gives comparable execution times for the preconditioned solver. Larger sparsity patterns for the FSAI preconditioners ($q = 2, 3$) do not perform well on the GPU due to less caching efficiency which results in bandwidth degradation in contrast to the CPU performance.

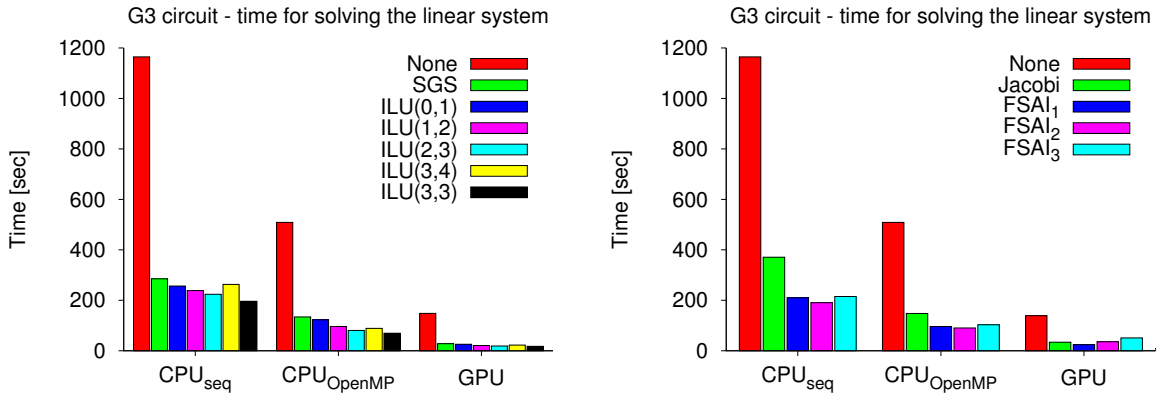

Figure 5.41: `g3_circuit` matrix: Performance benchmarks on the single/eight-core CPU and on the GPU. Solver time for the multi-colored SGS and the power(q)-pattern enhanced ILU(p,q) (left), Jacobi and FSAI algorithms based on $|A|^1$, $|A|^2$ and $|A|^3$ (right)

In Figure 5.41 the performance of the preconditioner for the `g3_circuit` matrix is shown. The figure shows the number of iterations with and without preconditioner. Between 4 and 25

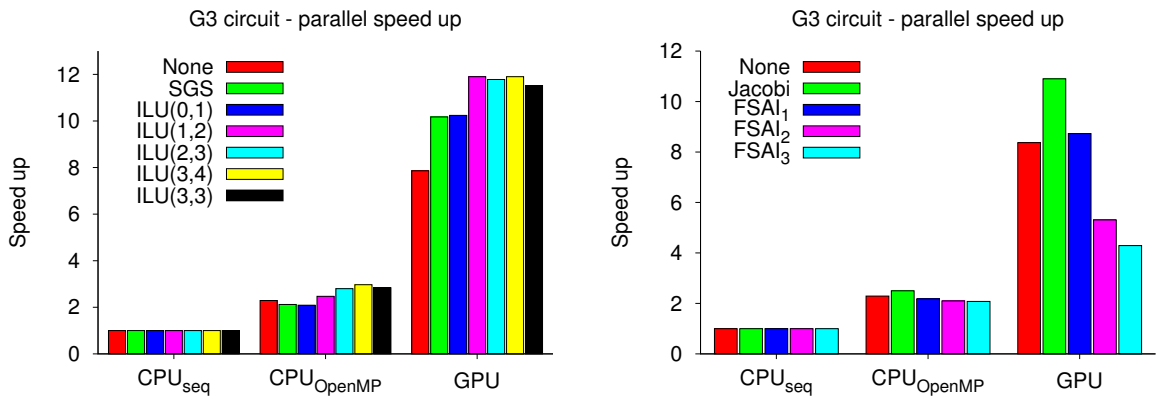

Figure 5.42: `g3_circuit` matrix: Parallel speed ups for various preconditioned CG solvers on the CPU OpenMP and the GPU platform

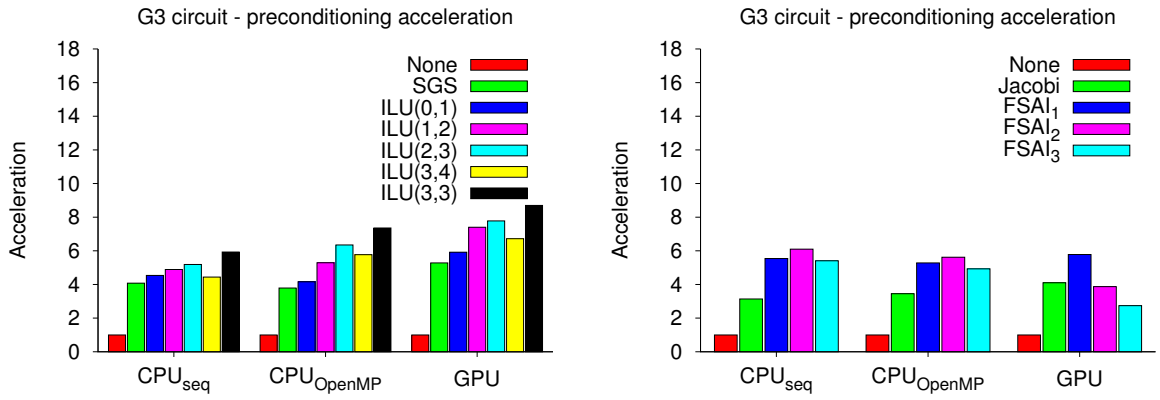

Figure 5.43: `g3_circuit` matrix: Acceleration factor of the preconditioning phase in the CG solver on the single/eight-core CPU and on the GPU platform

colors provides the multi-coloring decomposition for this problem. The multi-colored SGS and the multi-colored ILU(0,1) decompositions have 4 colors, whereas the power(q)-pattern method enhanced multi-colored ILU(3,4) has 35 colors. All the preconditioners show a significant decrease of the iteration count. In Figure 5.42 we find parallel speed ups between 2 and 3 for the eight core OpenMP parallel version and 8 to 12 for the data-parallel GPU version. Again, the GPU version is by a factor of 3 to 5 faster than the OpenMP parallel version on eight CPU cores. Figure 5.43 shows the acceleration factor of the preconditioner with respect to the time on a fixed hardware configuration and implementation. The total acceleration factor is between 4 and 9 for this test problem. Acceleration and parallel speed ups are observed for all presented implicit preconditioner configurations. The drop-off technique for ILU(3,3) reduces the number of colors from 35 to 17. It gives the best results with respect to the solver time and better acceleration factors. The explicit preconditioners show good scalability on the CPU, however on the GPU the larger sparsity patterns show again decreased performance.

In general, small matrices (like `3dkq4m2`) are better suited for the cache-oriented CPUs because sub-blocks or parts of the matrix and solution vectors can be kept in the cache and no further access to the main memory is necessary. Figure 5.44 and Figure 5.45 depict performance data for the `s3dkq4m2` matrix. Table 5.4 and Table 5.5 show that the number of iterations without preconditioner is massive but can be decreased considerably by preconditioning. Furthermore, we can see that the best results in terms of reduction of iterations are achieved for the ILU(3,4) scheme with 150 colors. But for this matrix, the best parallel solver times are obtained by the

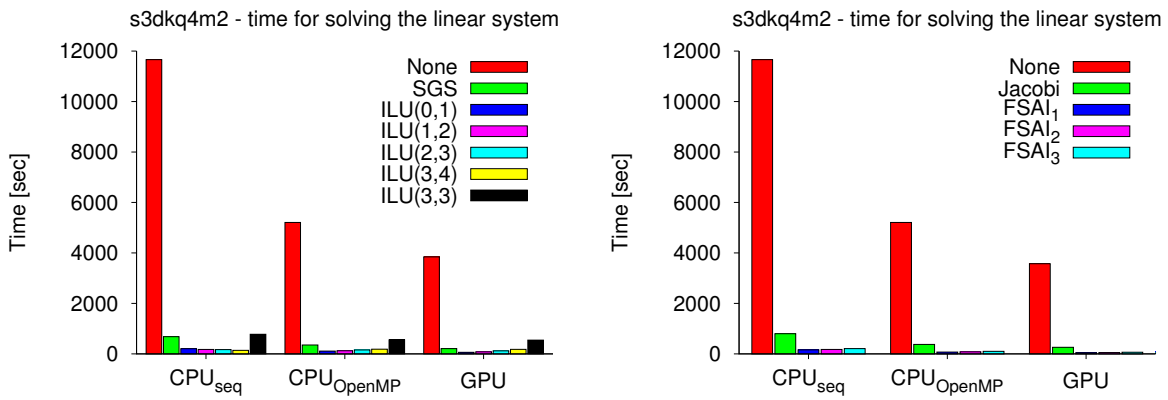

Figure 5.44: `s3dkq4m2` matrix: Performance benchmarks on the single/eight-core CPU and on the GPU. Solver time for the multi-colored symmetric SGS and the power(q)-pattern enhanced ILU(p,q) (left), Jacobi and FSAI algorithms based on $|A|^1$, $|A|^2$ and $|A|^3$ (right)

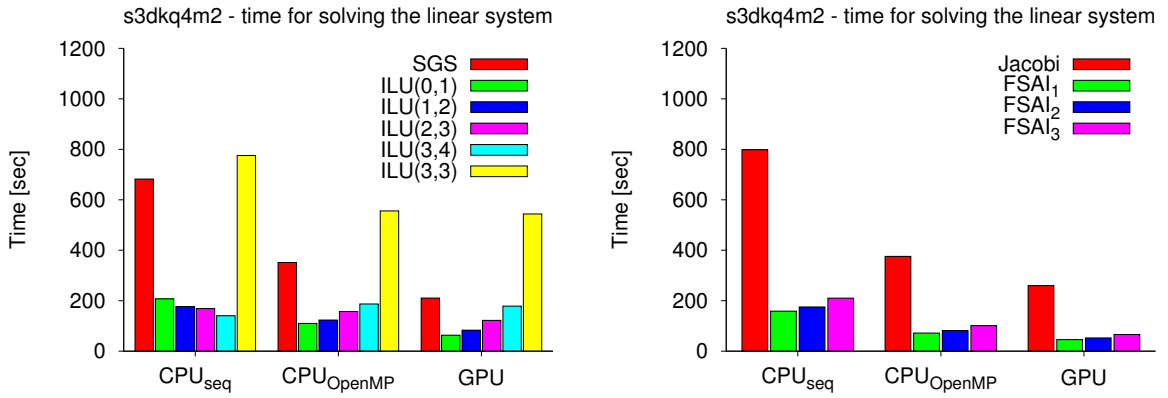

Figure 5.45: `s3dkq4m2` matrix: Zoomed-in performance benchmarks plots on the single/eight-core CPU and on the GPU, the non-preconditioned solver time is omitted. Solver time for the multi-colored SGS and the power(q)-pattern enhanced ILU(p,q) (left), Jacobi and FSAI algorithms based on $|A|^1$, $|A|^2$ and $|A|^3$ (right)

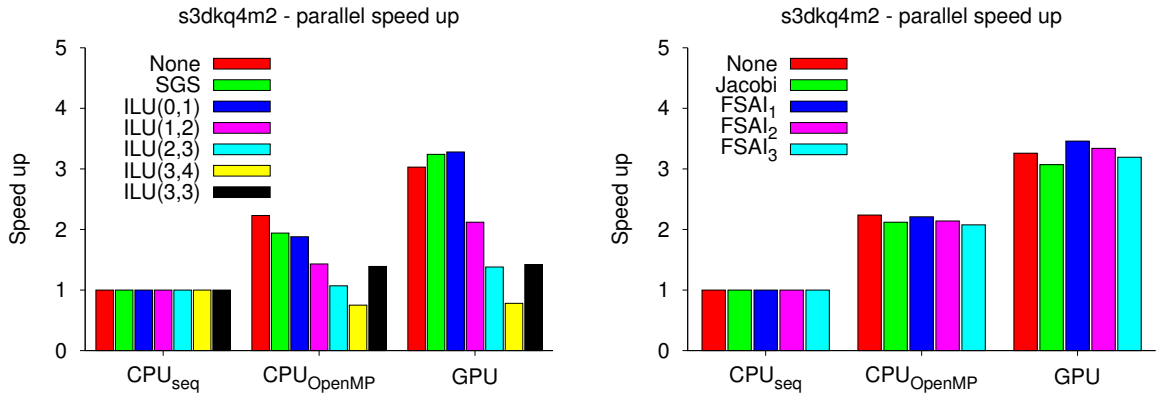

Figure 5.46: `s3dkq4m2` matrix: Parallel speed ups for various preconditioned CG solvers on the CPU OpenMP and the GPU platform

FSAI$_1$ preconditioner, see Figure 5.44 and Figure 5.45. For the power(q)-pattern method the drop-off technique reduces the number of colors from 134 to 96 for $p = 3$ but has no positive

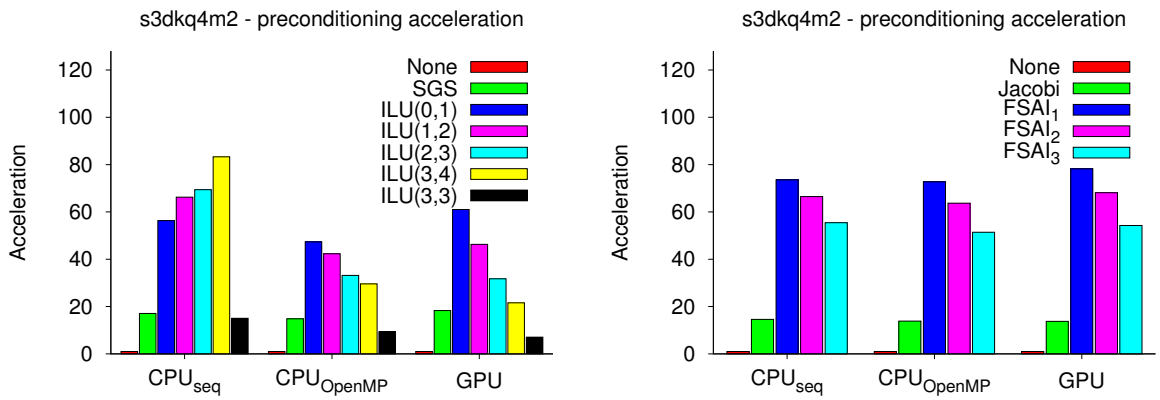

Figure 5.47: `s3dkq4m2` matrix: Acceleration factors for the preconditioning phase in the CG solver on the single/eight-core CPU and on the GPU platform

effect with respect to the performance efficiency. For the same preconditioner, the solver times are improved by the parallel OpenMP and GPU versions only for $p = 0, 1, 2$, with best results for $p = 0$. In Figure 5.46 is shown that for the power(q)-pattern method the OpenMP parallel speed up is not much more than two and gets worse when increasing p. In contrast to that the approximate inverse preconditioners scale well on all platforms. But the performance decreases with increasing the number of fill-in entries of the inverse preconditioner. The difference here are the forward and backward substitutions which require a large number of SpMV operations and thus this leads to a large overhead on the CPU and on the GPU. This is not the case for the approximate inverse which requires only one SpMV operation. For the implicit preconditioners, the maximal speed up on the GPU is around three and about 50% higher than the eight core OpenMP parallel speed up. Similar observations are made for the implicit preconditioners. In the parallel case, the acceleration factors from preconditioning are still immense, see Figure 5.47. Compared to the sequential case, the implicit preconditioner does not decrease much of its efficiency. For ILU(0,1) the acceleration factor is more than 60 on the GPU. However, for increasing p, results on the GPU and the parallel OpenMP version get worse while they get better in the sequential CPU case, because the sub-matrices contain only very few elements. For this test problem, the drop-off technique with a reduced number of colors in ILU(3,3) gives poor results. On the other hand, the inverse preconditioners provide the same acceleration profile on both platforms.

5.4.6 Comparison of Level-scheduling and Power(q)-pattern Method

To compare the performance of the level-scheduling and the power(q)-pattern method we follow [98]. We test the preconditioned CG and BiCGStab solvers using the power(q)-pattern method in lmpLAToolbox (see Sections 4), the level-scheduling in CUSPARSE library [102] and the Intel MKL library [73], and we compare the obtained results. Since the implementation of the preconditioning solvers is different, this gives us only a rough estimate of the performance run times. In 9 of 11 cases the lmpLAToolbox CPU version (power(q)-pattern method) is faster than the MKL-based solver, and in 10 of 11 it performs better than the CUSPARSE solver on the GPU. Details of the comparison can be seen in Appendix D.

5.4.7 Poisson Problem – CG Solver on GPU Cluster

In this last section, we demonstrate the scalability of the lmpLAToolbox on a heterogeneous cluster equipped with GPU devices. The simulations are performed on the TinyGPU cluster at the Regionales Rechenzentrum Erlangen (RRZE), Germany. The GPU cluster contains 8 nodes

with two Xeon 5500 CPUs. The system's interconnect is an Infiniband fabric with 20 Gbit/s bandwidth per link and direction. The network connectivity consists of a small 24-port DDR switch and it is used to provide full non-blocking communication. Each of the nodes include two NVIDIA Tesla M1060 GPU boards attached via two 16-lane PCIe bus connections (16 GPUs in total). All GPU boards are equipped with 4 GBytes GDDR3 memory and the computing capability of the GPU devices is 1.3.

For the scaling test we use a 3D Poisson equation on the unit cube with a Q1 finite element discretization. The local sub-matrices and sub-vectors are associated with the MPI processes on the nodes of the cluster, see Section 4.6. The total number of degrees of freedom in all configurations is 2.1 million. To solve the linear system, we use the non-preconditioned CG method with a relative stopping criterion of 10^{-14} for the residual and zero initial values. To reach this stopping criterion, the CG solver without preconditioning needs 510 iterations for all decomposition scenarios.

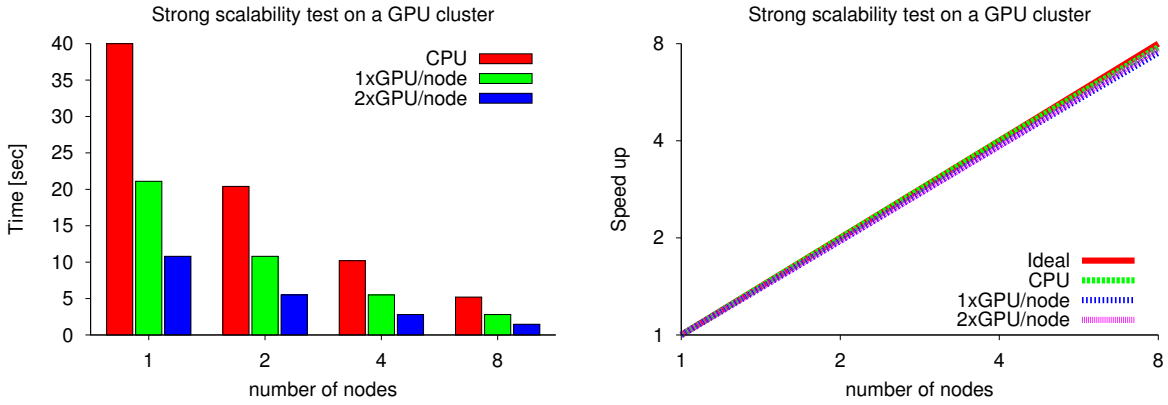

Figure 5.48: CPU and single/double GPU performance on an 8-node cluster; strong scalability test (left) and speed up factors (right) for the CG method applied to the 3D Poisson problem with 2.1 million DOF for three cluster configurations

Figure 5.48 (left) depicts the total run time for the CG solver with respect to the number of used nodes (1 to 8) in the GPU-accelerated cluster. It details the results for the sequential CPU version, the configuration with a single GPU per node and with two GPUs per node. The single GPU configuration runs approximately twice as fast as the CPU-only configuration. The dual GPU case gives a speed up of an additional factor of approximately two. To utilize both GPUs of a node we assign two processes to each node. Due to an optimized geometric partitioning of the mesh we ensure that the communication during the solution procedure is kept at a minimum, see [4, 57]. As described in Section 4, irregular accesses of data buffers for the ghost DOF within the SpMV is pre-processed on the CPU and sent in bulk chunks over the PCIe bus. Due to the advanced data handling technique, the performance ratio of the single GPU per node configuration and the dual GPU per node configuration (i.e. 8 nodes, 16 GPUs) is approximately 1.9.

The speed up for the strong scalability test for all three configurations is presented in Figure 5.48 (right). On eight nodes the speed up with respect to a single node is very close to the optimal value of eight.

Chapter 6

Summary and Further Work

Multi-core and many-core devices provide high performance capability. In order to exploit this potential many numerical algorithms and schemes need to be adapted. The algorithms and their implementations need to have fine-grained parallelism in order to fully utilize the modern parallel devices.

The main focus of this work was to build efficient parallel iterative solvers for large and sparse linear systems for multi-core and many-core devices. We have shown that after an adaption of the numerical schemes with respect to the fine-grained parallelism we are able to harness the potential of modern parallel platforms. We have demonstrated that Krylov subspace methods and multi-grid methods combined with highly parallel preconditioners and smoothers lead to efficient utilization of these new technologies. This work provides the next step in the field of iterative solvers for many-core systems in terms of new numerical algorithms with fine-grained parallelism.

We started with an introduction to the basics of the FEM for linear elliptic PDEs, presented in Chapter 2. We continued with remarks on linear solvers and we focused on iterative solvers including simple iteration schemes, Krylov subspaces solvers and geometric multi-grid methods. We emphasized the importance of proper preconditioning/smoothing techniques and gave a brief overview of the existing schemes. We concluded this chapter with remarks on the algorithmic complexity and computational intensity.

In Chapter 3, we showed the way the basic routines can be executed in parallel and we made remarks on the accuracy and the consistency of the results. We proceeded by presenting our concept for full-parallel and hybrid-parallel solvers on multi-core and many-core devices. The main focus in this chapter was on the parallel preconditioners – here, we demonstrated how to apply additive, multiplicative and approximate inverse preconditioners in parallel. For LU-based preconditioners we proposed a block-wise form of the forward and backward substitutions. It is based on vector/matrix operations and therefore can be performed in parallel. Here, we reviewed two techniques – multi-coloring and level-scheduling re-ordering algorithms. We proposed a novel method, called the *power(q)-pattern method*, for controlling the sparsity pattern of the ILU decomposition with fill-ins based on levels. We proved that this algorithm can be used to produce a new matrix structure with diagonal blocks containing only diagonal entries. With this approach we obtained a higher degree of parallelism in comparison with the level-scheduling algorithm. At the end of this chapter we presented remarks on other parallel preconditioning schemes.

We progressed with a software design of a library for sparse linear solvers which was presented in Chapter 4. In this section, we gave a description of the LAToolbox in the HiFlow^3 project. We elaborated our concept for developing a multi-platform linear algebra toolbox library. Using the proposed model for memory accesses and key object methods, we were able to build solvers and preconditioners for various backends while maintaining a single source code.

In the last part (Chapter 5 and Appendices B, C, D) we conducted numerical experiments

and we provided benchmarks, evaluation and performance analysis over various hardware and linear systems. In this section, we considered Krylov space solvers (CG, BiCGStab, GMRES) and matrix-based geometric multi-grid methods (V and W-cycles) combined with different preconditioners and smoothers. We demonstrated the power and the efficiency of the parallel preconditioners in terms of acceleration factors – reduction of the computational time by deploying a preconditioning phase. Furthermore, we showed the speed ups when performing the solvers on various parallel systems.

In this thesis, we have focused on fine-grained parallel techniques for performing additive, multiplicative and approximate inverse proconditioners and on their embedding in iterative linear solvers. As next steps, we need to adapt and explore other preconditioning schemes and solvers such as support-tree based preconditioners and algebraic multi-grid methods. Furthermore, we need to focus on the pre-processing and on the post-processing steps of the solvers and of the preconditioners. In particular, by providing a parallel implementation of the building phase for the preconditioning equation we will have better insights of the overall performance of the preconditioned parallel solvers.

Appendix A

Source Code Examples for Preconditioned CG

Here we present a source code for preconditioned CG solver based on the local multi-platform linear algebra toolbox.

Listing A.1: CG solver implementation on local multi-platform linear algebra toolbox (lmpLA-Toolbox)

```
1  // Preconditioned CG
2  template <typename ValueType>
3  void cg(lVector<ValueType> *x, lVector<ValueType> *b,
4          lMatrix<ValueType> *matrix,
5          ValueType eps, int max_iter,
6          lPreconditioner<ValueType> *lPrecond)
7  {
8    int iter;
9    lVector<ValueType> *p, *q, *r, *z ;
10   ValueType rho, rho_old, alpha, beta, start_norm, r_norm ;
11
12   x->print();
13   b->print();
14   matrix->print();
15
16   p = x->CloneWithoutContent() ;
17   q = x->CloneWithoutContent() ;
18   r = x->CloneWithoutContent() ;
19   z = x->CloneWithoutContent() ;
20
21   rho = static_cast<ValueType> (0.0) ;
22
23   matrix->VectorMult(*x, r) ;
24   r->ScaleAdd( static_cast<ValueType> (-1.0), *b);   ; // r = b - Ax
25
26   rho_old = rho ;
27
28   lPrecond->ApplylPreconditioner(*r, z);
29   rho = r->Dot(*z) ;
30
31   lPrecond->print();
```

```
32
33    start_norm = r->Nrm2() ;
34
35    for (iter=0; iter<max_iter; ++iter) {
36
37        lPrecond->ApplylPreconditioner(*r, z);
38
39        rho_old = rho ;
40        rho = r->Dot(*z) ;
41        beta = rho/rho_old ;
42
43        if (iter == 0)
44          beta = static_cast<ValueType> (0.0) ;
45
46
47        p->ScaleAdd(beta, *z) ; //p=beta*p + z
48        matrix->VectorMult(*p, q) ;
49        alpha = rho / (p->Dot(*q)) ;
50        x->Axpy(*p, alpha);
51        r->Axpy(*q, (static_cast<ValueType> (-1.0))*alpha);
52
53        r_norm = r->Nrm2() ;
54
55        if ((r_norm <= eps*start_norm) ||
56            (r_norm <= ABS_EPS) )
57          break;
58
59    }
60
61    std::cout << "iterations="
62              << iter << "; L2 res="
63              << r->Nrm2()
64              << std::endl << std::endl;
65
66    delete p;
67    delete q;
68    delete r;
69    delete z;
70
71 }
```

Appendix B

Complimentary SPD Matrix Tests

In this appendix we present a qualitative comparison between several preconditioning schemes on real, symmetric and positive definite matrices. For the tests, we use preconditioned CG solver with a zero initial value and a right-hand side equal to one. For the stopping criterion we use relative and absolute stopping factors equal to 10^{-6} or 10^{-16}. All of the tests are performed in double precision. In Table B.1, B.2 B.3 the number of iterations and the level of parallelism (e. g. number of colors and number of levels for LU-sweeps), as well as the number of non-zeroes for approximate inverse matrix based on FSAI algorithm, $\mathcal{NNZ} = \mathcal{NNZ}(L) + \mathcal{NNZ}(L^T)$ are shown. With 'no conv' we denote the inability of the solver to finish in 100,000 iterations.

		gr3030 [86]	bcsstk06 [82]	bcsstk13 [83]	bcsstk14 [84]	bcsstk15 [85]	mhd3200b [128]	nos1 [87]
No Precond	# iter	33	3990	342923	14012	22016	305293	2107
SGS	# iter	24	183	548	196	213	17	230
	# colors	4	15	41	25	19	6	2
ILU(0,1)	# iter	23	175	14795	98	1124	10	142
	# colors	4	15	41	25	19	6	2
ILU(1,2)	# iter	16	85	2184	42	160	5	86
	# colors	9	35	137	65	77	10	6
ILU(2,3)	# iter	12	50	222	26	71	4	50
	# colors	16	59	321	122	168	14	8
ILU(3,4)	# iter	10	19	59	20	46	3	37
	# colors	26	91	589	192	270	18	10
ILU(3,3)	# iter	11	44	73	19	51	4	64
	# colors	16	59	321	122	168	14	8
ILU(0)	# iter	16	43	no conv	250	544	2	9183
	# levels	88	88	577	392	365	798	79
ILU(1)	# iter	11	20	159	31	75	0	0
	# levels	117	146	10000	582	844	798	157
ILU(2)	# iter	8	15	113	20	55	0	0
	# levels	146	202	1370	797	1254	798	157
ILU(3)	# iter	7	10	51	16	41	0	0
	# levels	175	256	1749	962	1649	798	157

Table B.1: Qualitative comparison between several parallel preconditioners for seven test matrices – level-scheduling and power(q)-pattern method

		gr3030 [86]	bcsstk06 [82]	bcsstk13 [83]	bcsstk14 [84]	bcsstk15 [85]	mhd3200b [128]	nos1 [87]
Jacobi	# iter	33	409	1449	409	575	36	450
$FSAI_1$	# iter	25	159	515	95	220	10	252
	$\mathcal{NNZ}$	8644	8280	85886	65260	121764	21516	1254
$FSAI_2$	# iter	17	99	260	55	133	6	129
	$\mathcal{NNZ}$	21636	24080	398776	197460	531636	37344	2182
$FSAI_3$	# iter	13	83	122	39	92	4	85
	$\mathcal{NNZ}$	40104	44250	954444	380234	1129186	53096	2942

Table B.2: Qualitative comparison between several parallel preconditioners for seven test matrices – Jacobi and FSAI algorithm

		mesh3e1 [127]	s1rmt3m1 [94]	nos2 [88]	nos3 [89]	nos4 [90]	nos5 [91]	nos6 [92]	nos7 [93]
No Precond	# iter	17	6722	36466	229	75	427	1001	3375
SGS	# iter	7	293	2967	83	33	114	47	43
	# colors	5	24	2	10	4	9	2	2
ILU(0,1)	# iter	6	262	1617	80	31	99	46	40
	# colors	5	24	2	10	4	9	2	2
ILU(1,2)	# iter	4	156	864	49	18	30	25	25
	# colors	11	54	6	18	13	33	7	12
ILU(2,3)	# iter	3	107	486	36	15	18	21	17
	# colors	20	96	8	32	18	84	8	21
ILU(3,4)	# iter	2	79	360	26	11	9	17	14
	# colors	30	153	10	50	31	157	18	41
ILU(3,3)	# iter	3	86	660	29	12	11	22	15
	# colors	20	96	8	32	18	84	8	21
ILU(0)	# iter	4	179	no conv	43	19	40	27	25
	# levels	71	355	319	132	32	40	44	25
ILU(1)	# iter	2	83	0	29	12	22	18	19
	# levels	72	532	637	178	43	137	73	49
ILU(2)	# iter	1	49	0	22	7	13	16	15
	# levels	102	709	637	224	70	313	102	89
ILU(3)	# iter	1	32	0	18	6	6	12	11
	# levels	102	886	637	270	79	404	131	145
Jacobi	# iter	13	842	5805	203	67	234	94	87
$FSAI_1$	# iter	8	302	2897	95	37	117	47	42
	$\mathcal{NNZ}$	1666	223140	5094	16804	694	5640	3930	5346
$FSAI_2$	# iter	5	197	1347	63	25	57	32	30
	$\mathcal{NNZ}$	3710	584524	8902	43824	1902	27390	8856	15660
$FSAI_3$	# iter	3	145	874	46	19	34	25	23
	$\mathcal{NNZ}$	6582	1099738	12062	80832	3412	69562	15900	34768

Table B.3: Qualitative comparison between several parallel preconditioners for eight test matrices – level-scheduling and power(q)-pattern method; Jacobi and FSAI algorithm

Appendix C

Complimentary Performance Tests on Intel i7-2600 and NVIDIA GTX580

In this appendix we present benchmark performance tests on a single-socket Intel i7-2600 quad-core system (with Hyper-Threading) which is accelerated by an NVIDIA GeForce GTX 580 device with memory capacity of the CPU system and GPU device is 16GB and 3GB respectively. Due to the Hyper-Threading on the CPU, we run the OpenMP tests with eight threads. Note that some of the problems are not fitting into the 3GB memory of the GPU and bars are left blank.

C.1 Poisson Problem - Multi-grid and CG Solver

Here we present the solution times for multi-grid solver and CG solver (see Section 5.2.3,5.3.5 and Section 5.4.3).

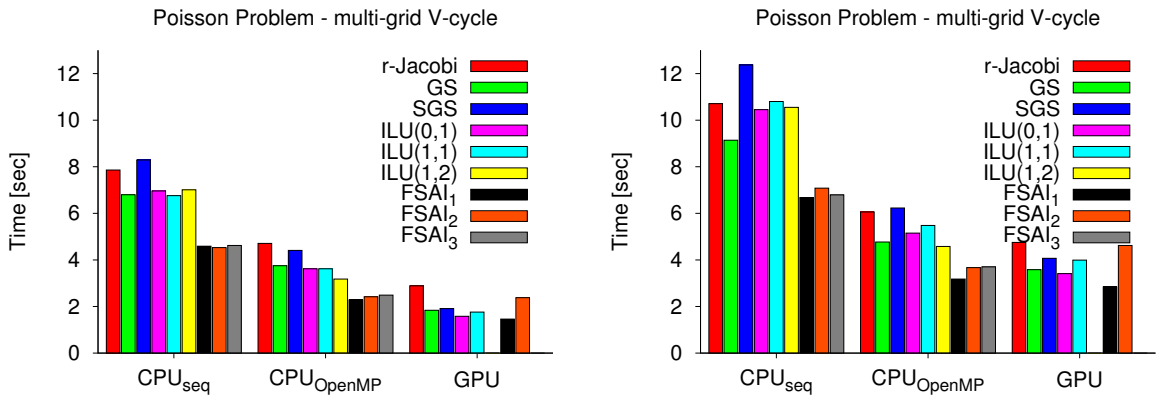

Figure C.1: Run time for the multi-grid V-cycle (left) and W-cycle (right) for the Poisson problem on the 2D L-shaped domain

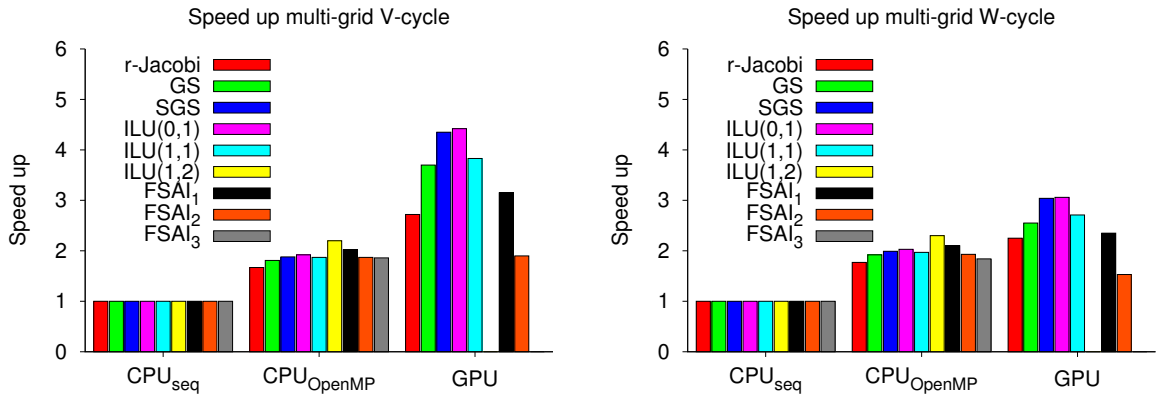

Figure C.2: Parallel speed ups of the multi-grid V-cycle (left) and W-cycle (right) for the Poisson problem on the 2D L-shaped domain

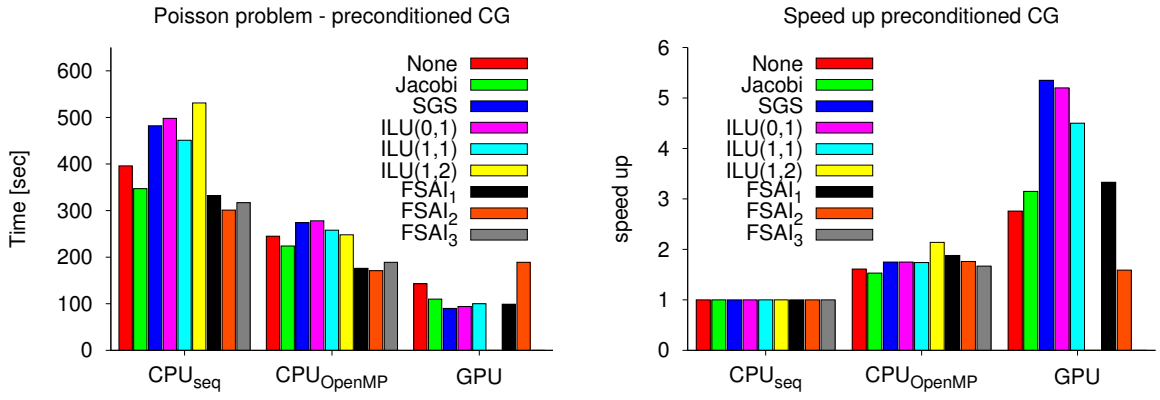

Figure C.3: Run time performance and parallel speed up factors of the preconditioned CG solver for the Poisson problem on the 2D L-shaped domain

C.2 Matrix Collection Problems - CG Solver

Here we present the performance profile for three matrices (see Section 5.1.3) based on various preconditioned CG (see Section 5.2).

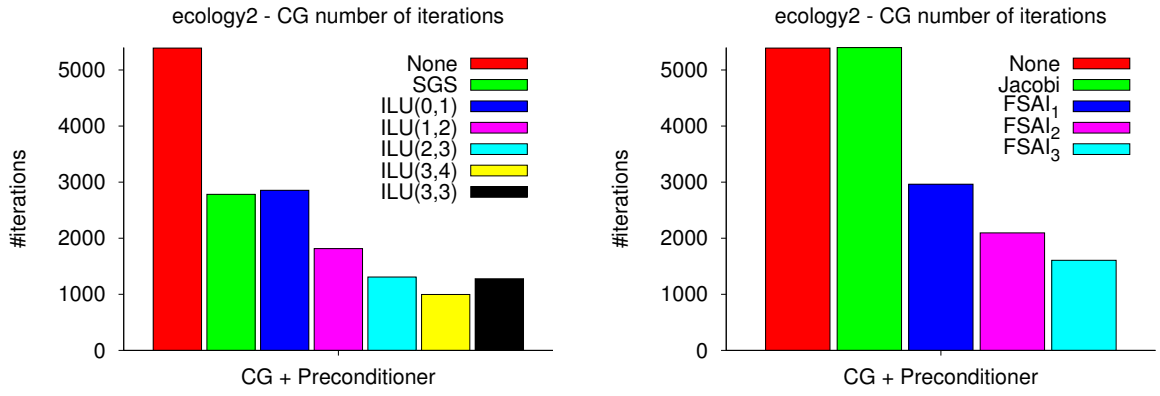

Figure C.4: `ecology2 matrix`: Number of iterations for solving the linear system with the preconditioned CG solver based on various preconditioning schemes

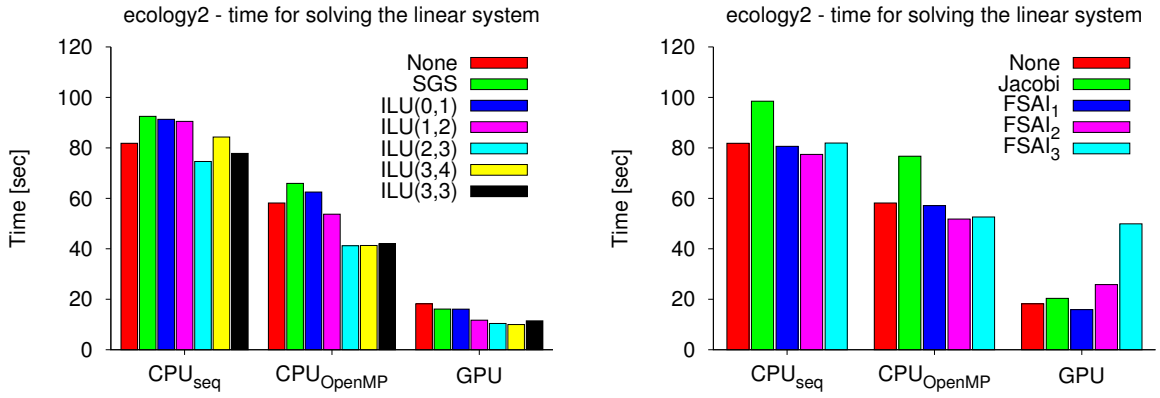

Figure C.5: `ecology2 matrix`: Performance benchmarks on the single/four-core CPU and on the GPU. Solver time for the multi-colored SGS and the power(q)-pattern enhanced ILU(p,q) (left), Jacobi and FSAI algorithms based on $|A|^1$, $|A|^2$ and $|A|^3$ (right)

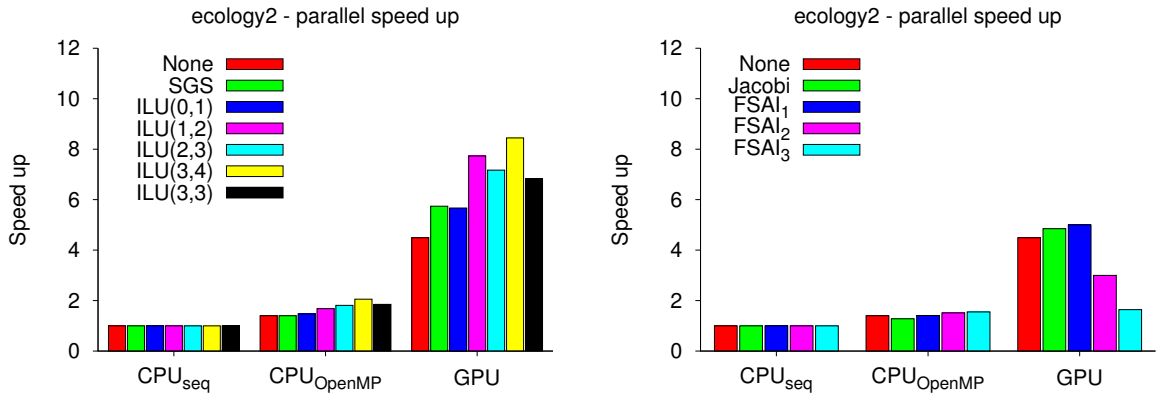

Figure C.6: `ecology2` matrix: Parallel speed ups for various preconditioned CG solvers on the CPU OpenMP and the GPU platform

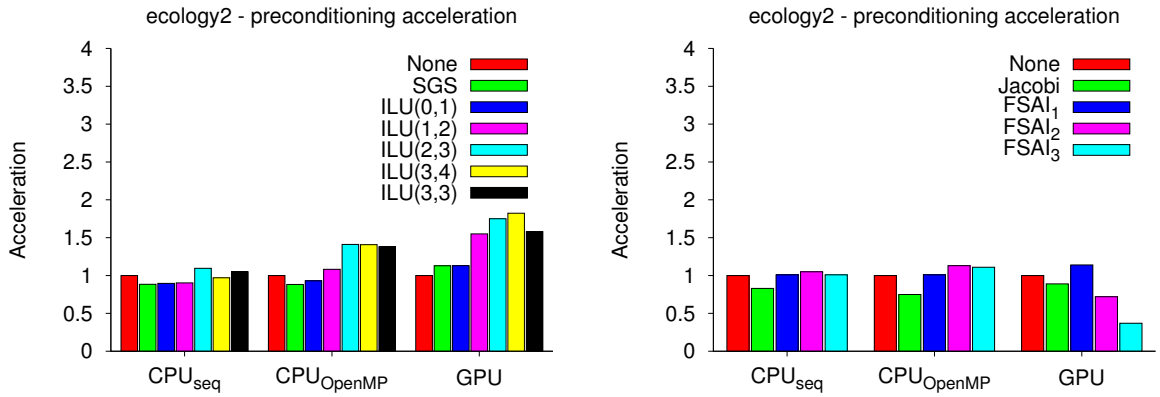

Figure C.7: `ecology2` matrix: Acceleration factors for the preconditioning phase in the CG solver on the single/four-core CPU and on the GPU platform

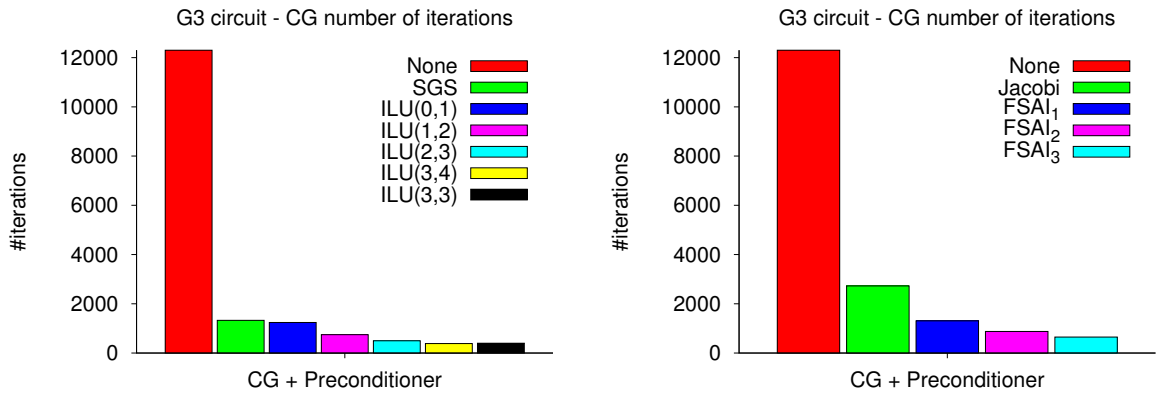

Figure C.8: `g3_circuit` matrix: Number of iterations for solving the linear system with the preconditioned CG solver based on various preconditioning schemes

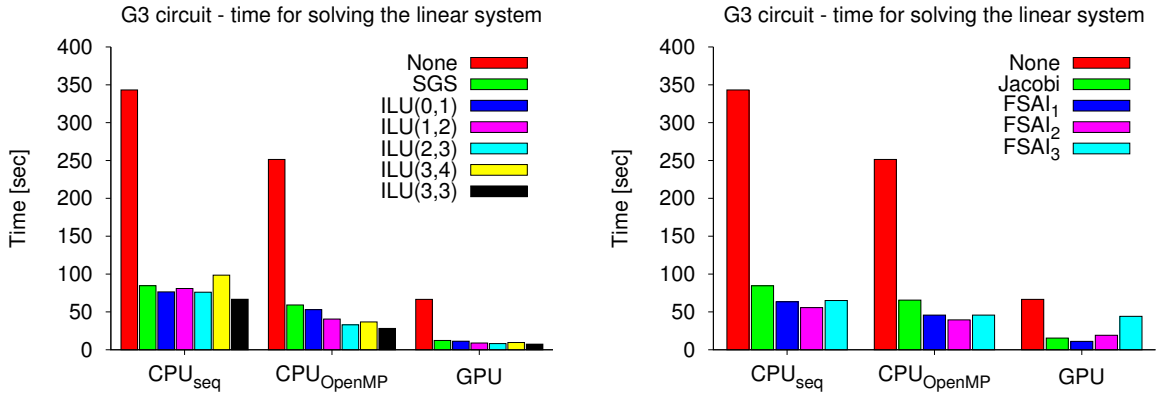

Figure C.9: `g3_circuit` matrix: Performance benchmarks on the single/four-core CPU and on the GPU. Solver time for the -colored SGS and the power(q)-pattern enhanced ILU(p,q) (left), Jacobi and FSAI algorithms based on $|A|^1$, $|A|^2$ and $|A|^3$ (right)

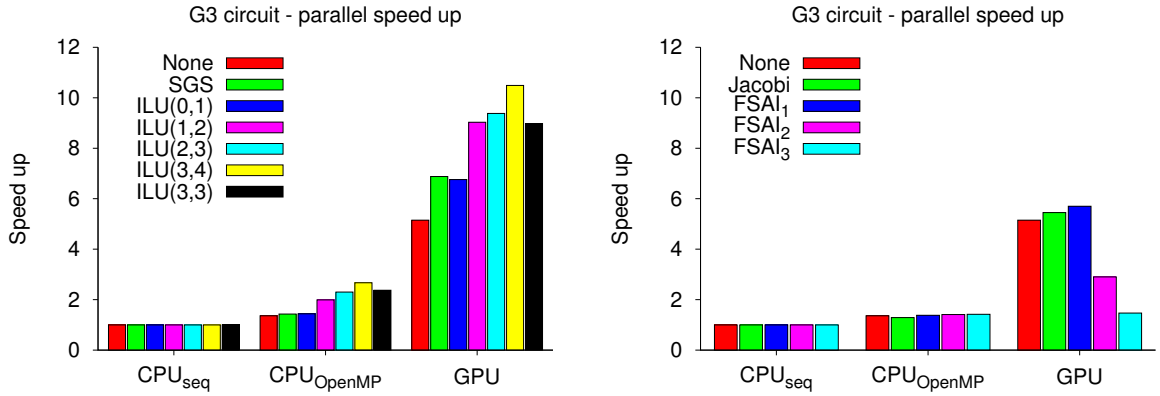

Figure C.10: `g3_circuit` matrix: Parallel speed ups for various preconditioned CG solvers on the CPU OpenMP and the GPU platform

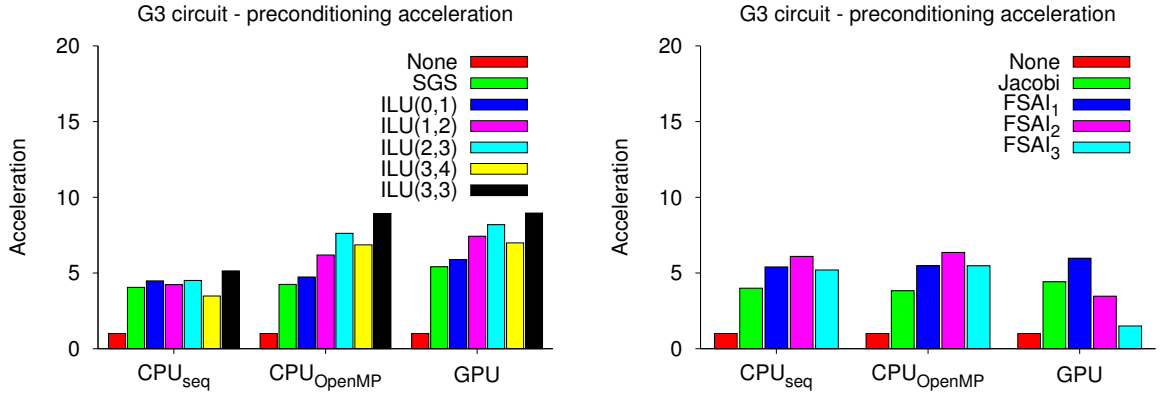

Figure C.11: `g3_circuit` matrix: Acceleration factors for the preconditioning phase in the CG solver on the single/four-core CPU and on the GPU platform

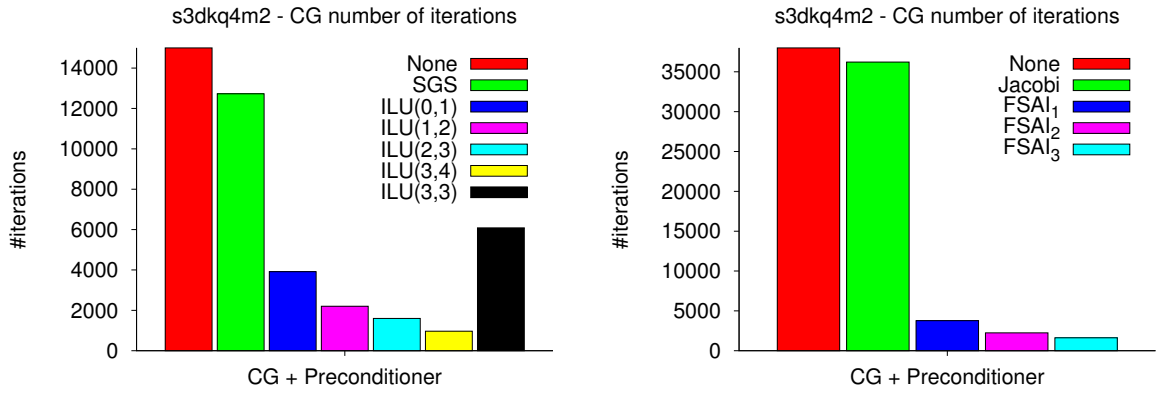

Figure C.12: `s3dkq4m2` matrix: Number of iterations for solving the linear system with the preconditioned CG solver based on various preconditioning schemes

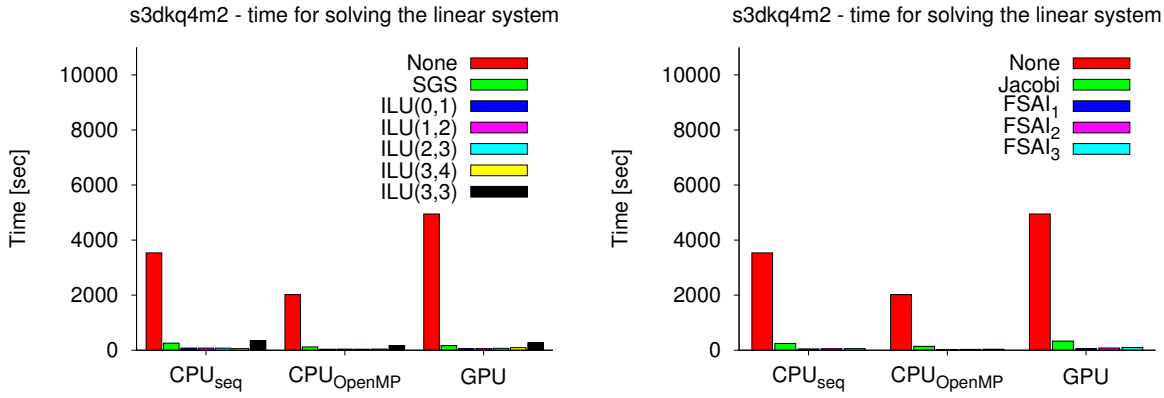

Figure C.13: `s3dkq4m2` matrix: Performance benchmarks on the single/four-core CPU and on the GPU. Solver time for the multi-colored SGS and the power(q)-pattern enhanced ILU(p,q) (left), Jacobi and FSAI algorithms based on $|A|^1$, $|A|^2$ and $|A|^3$ (right)

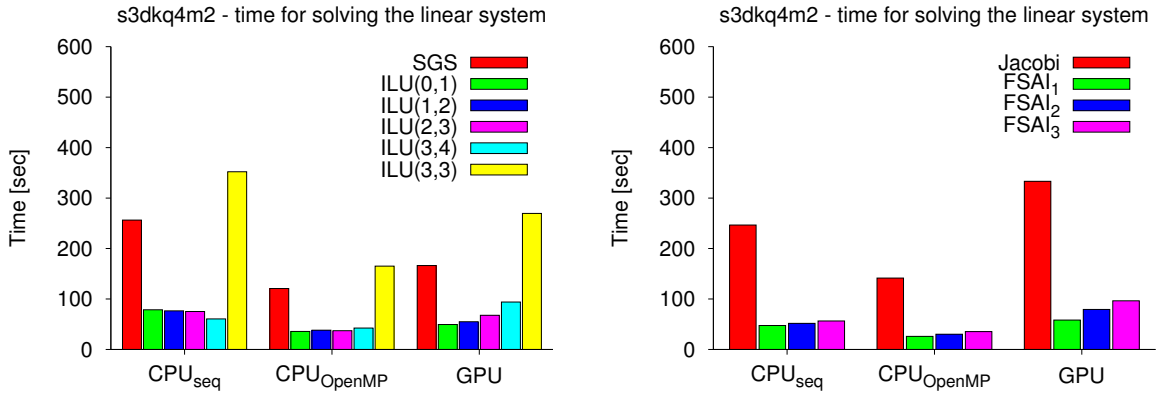

Figure C.14: `s3dkq4m2` matrix: Zoomed-in performance benchmarks plots on the single/four-core CPU and on the GPU, the non-preconditioned solver time is omitted. Solver time for the multi-colored SGS and the power(q)-pattern enhanced ILU(p,q) (left), Jacobi and FSAI algorithms based on $|A|^1$, $|A|^2$ and $|A|^3$ (right)

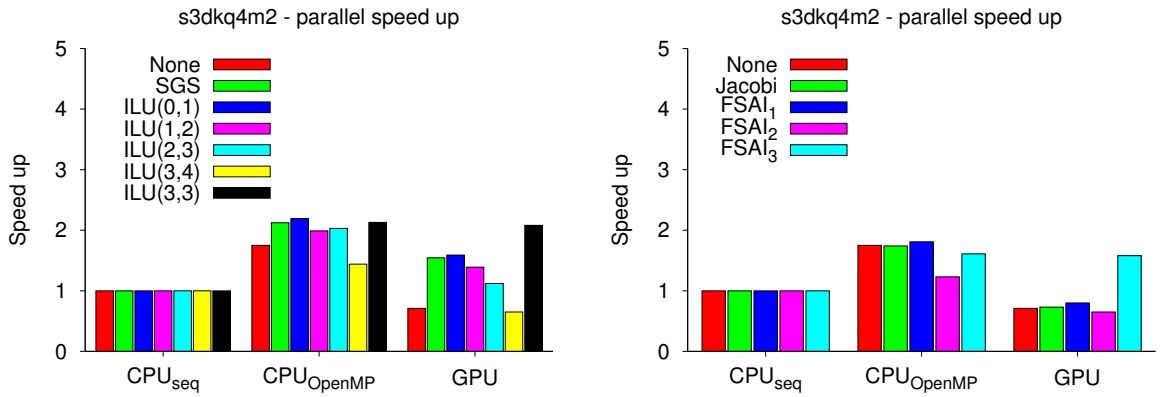

Figure C.15: `s3dkq4m2` matrix: Parallel speed ups for various preconditioned CG solvers on the CPU OpenMP and on the GPU platform

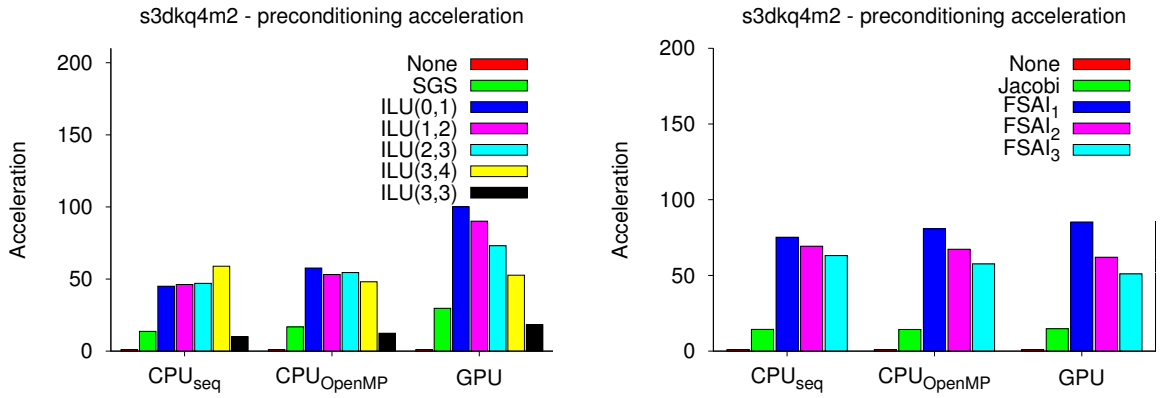

Figure C.16: `s3dkq4m2` matrix: Acceleration factors for the preconditioning phase in the CG solver on the single/four-core CPU and on the GPU platform

Appendix D

Performance Comparison of Level-scheduling and Power(q)-pattern Method

In order to do a comparison of the level-scheduling and the power(q)-pattern method, we follow the work of [98]. For the numerical experiments we use a zero initial value and a right-hand side $b = Ae$, where e is a vector filled with ones. For the solving phase we apply CG and BiCGStab solvers with a relative stopping criterion of 10^{-7}. The comparison is based on the most scalable version (i.e. ILU(0)) of the CUSPARSE against the MKL and the equivalent to the power(q)-pattern method – ILU(0,1). The selected matrices are presented in Table D.1, they are obtained from the University of Florida Sparse Matrix Collection [129].

	Matrix	Size	NNZ	spd
1	offshore	259,789	4,242,673	yes
2	af_shell3	504,855	17,562,051	yes
3	parabolic_fem	525,825	3,674,625	yes
4	apache2	715,176	4,817,870	yes
5	ecology2	999,999	4,995,991	yes
6	thermal2	1,228,045	8,580,313	yes
7	G3_circuit	1,585,478	7,660,826	yes
8	FEM_3D_thermal2	147,900	3,489,300	no
9	ASIC_320ks	321,671	1,316,085	no
10	cage13	445,315	7,479,343	no
11	atmosmodd	1,270,432	8,814,880	no

Table D.1: Symmetric and non-symmetric test matrices

The comparison gives only a rough estimation of the performance profile, presented in Table D.2. One of the reasons is that we measure the total time for execution of the CG method (see `cg` function in Appendix A), including some memory allocation inside the function. This changes the performance profile for very small iteration numbers as for matrix 1, 8, 9 and 10. Second, the lmpLAToolbox SpMV function is not as efficient as the MKL nor the CUSPARSE routine. Third, the lmpLAToolbox LU-based preconditioning solver is based on explicit SpMV multiplication for each block (see Chapter 3), however this is not for the CUSPARSE routine, see [98]. In other words, the implementation of the level-scheduling and the power(q)-pattern method are different, thus the comparison of the two methods are not very fair.

The power(q)-pattern method runs on an Intel Core i7 920 CPU which is clocked at 2.67 GHz, the data is presented in Table D.2 as numhpc0327. For the MKL benchmarks, an Intel Core i7 950 CPU clocked at 3.07GHz has been used. Both CPUs are based on the same core

matrix	time [sec] #iter	ILU(0) MKL	ILU(0,1) numhpc0327	ILU(0) CUSPARSE	ILU(0,1) GTX580	ILU(0,1) S1070
1	time	0.72	0.47	1.52	0.36	0.55
	#iter	25	32	25	32	32
2	time	38.5	49.1	33.9	37.2	41.5
	#iter	569	939	571	938	939
3	time	39.2	19.2	6.91	5.80	12.6
	#iter	1099	1134	1099	1133	1134
4	time	35.0	29.5	12.8	10.2	22.1
	#iter	713	1580	713	1579	1580
5	time	107	60.6	55.3	16.6	38
	#iter	1746	2889	1746	2888	2889
6	time	155	85.2	54.4	23.9	49
	#iter	1656	1748	1656	1747	1748
7	time	20.2	15.5	8.61	3.1	7.1
	#iter	183	338	183	338	338
8	time	0.13	0.17	0.52	0.17	0.23
	#iter	9	10	9	10	10
9	time	0.27	0.12	0.12	0.06	0.19
	#iter	6	5	6	5	5
10	time	0.28	0.17	0.15	0.11	0.17
	#iter	2.5	2	2.5	2	2
11	time	12.5	0.9	9.3	2.8	6.8
	#iter	76.5	125	79.5	120	135

Table D.2: Comparison of the ILU(0) with level-scheduling and the power(q)-pattern method for ILU(0,1)

architecture, known as Bloomfield. Apart from the higher clock rate, the CPUs are identical, which also includes the QPI bandwidth as well as the integrated memory controller.

The NVIDIA GTX580 with its GF110 chip consists of 512 Shader Processors (SPs). The core is clocked with 772 MHz, while the shader units are clocked at 1544 MHz. Its 3 GBytes GDDR5 memory is clocked at 1002 MHz, connected via a 384 bit bus width. This leads to a theoretical memory bandwidth of 192.4 GByte/sec. The Tesla S1070 system consists of four GPU devices, for this test we use only a single card. The cards are build on GT200 chips with 240 shader processors working on clock frequency of 602 MHz while the shaders operate at 1296 MHz. The device has access to a 4 GBytes GDDR3 memory, clocked at 800 MHz which is connected by a 512 bit interface. This results in a theoretical bandwidth of 102 GByte/sec. The GPU used for the CUSPARSE benchmarks, a NVIDIA C2050 (ECC on), is based on the Fermi chip GF100. It consists of 448 SPs with a core clock of 575 MHz and a shader clock of 1150 MHz. The 384 bit width bus interface connects 3072 MB of GDDR5 memory with the GPU. With the memory clock operating at 1500 MHz, a theoretical memory bandwidth of 144 GBytes is given.

Bibliography

[1] AMD. AMD Core Math Library (ACML) http://developer.amd.com/libraries/acml/, 2011.

[2] Amestoy, P. R., Duff, I. S., Koster, J., and L'Excellent, J.-Y. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications 23*, 1 (2001), 15–41.

[3] Amestoy, P. R., Guermouche, A., L'Excellent, J.-Y., and Pralet, S. Hybrid Scheduling for the Parallel Solution of Linear Systems. *Parallel Computing 32*, 2 (2006), 136–156.

[4] Anzt, H., Augustin, W., Baumann, M., Bockelmann, H., Gengenbach, T., Hahn, T., Heuveline, V., Ketelaer, E., Lukarski, D., Otzen, A., Ritterbusch, S., Rocker, B., Ronnas, S., Schick, M., Subramanian, C., Weiss, J.-P., and Wilhelm, F. HiFlow^3 – A Flexible and Hardware-Aware Parallel Finite Element Package http://www.emcl.kit.edu/preprints/emcl-preprint-2010-06.pdf. EMCL Preprint Series, 2010.

[5] Anzt, H., Hahn, T., Heuveline, V., and Rocker, B. GPU Accelerated Scientific Computing: Evaluation of the NVIDIA Fermi Architecture; Elementary Kernels and Linear Solvers http://www.emcl.kit.edu/preprints/emcl-preprint-2010-04.pdf. EMCL Preprint Series, 2010.

[6] Anzt, H., Rocker, B., and Heuveline, V. An Error Correction Solver for Linear Systems: Evaluation of Mixed Precision Implementations http://www.emcl.kit.edu/preprints/emcl-preprint-2010-01.pdf. EMCL Preprint Series, 2010.

[7] Anzt, H., Rocker, B., and Heuveline, V. Energy Efficiency of Mixed Precision Iterative Refinement Methods Using Hybrid Hardware Platforms: An Evaluation of Different Solver and Hardware Configurations http://www.emcl.kit.edu/preprints/emcl-preprint-2010-03.pdf. EMCL Preprint Series, 2010.

[8] Anzt, H., Rocker, B., and Heuveline, V. Mixed Precision Error Correction Methods for Linear Systems: Convergence Analysis based on Krylov Subspace Methods http://www.emcl.kit.edu/preprints/emcl-preprint-2010-02.pdf. EMCL Preprint Series, 2010.

[9] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. The Landscape of Parallel Computing Research: A View from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[10] ATLAS. Automatically Tuned Linear Algebra Software http://math-atlas.sourceforge.net/, 2011.

[11] Axelsson, O., and Barker, V. A. *Finite Element Solution of Boundary Value Problems: Theory and Computation.* Computer Science and Applied Mathematics. Acad. Pr., Orlando, 1984.

[12] Axelsson, O., and Vassilevski, P. Variable-step Multilevel Preconditioning Methods I: Self-adjoint and Positive Definite Elliptic Problems. *Numer. Linear Algebra Appl. 1*, 1 (1994), 75–101.

[13] Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.

[14] Bank, R. E., and Douglas, C. C. Sparse Matrix Multiplication Package (SMMP). *Advances in Computational Mathematics 1*, 1 (1993), 127–137.

[15] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2 ed. SIAM, Philadelphia, PA, 1994.

[16] Baskaran, M. M., and Bordawekar, R. Optimizing Sparse Matrix-vector Multiplication on GPUs. Tech. rep., IBM, 2009.

[17] Bebendorf, M. Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems, 2008.

[18] Beckermann, B., and Kuijlaars, A. B. J. Superlinear Convergence of Conjugate Gradients. *SIAM J. Numer. Anal 39* (1999), 300–329.

[19] Bell, N., and Garland, M. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *SC '09: Proc. of the Conf. on High Perf. Computing Networking, Storage and Analysis* (2009), ACM, pp. 1–11.

[20] Benzi, M., Meyer, C., and Tuma, M. A Sparse Approximate Inverse Preconditioner for the Conjugate Gradient Method. *SIAM J. Sci. Comput. 17*, 5 (1996), 1135–1149.

[21] Benzi, M., and Tuma, M. A Sparse Approximate Inverse Preconditioner for Nonsymmetric Linear Systems. *SIAM J. Sci. Comput. 19*, 3 (1998), 968–994.

[22] Berkeley Lab. UPC `http://upc.lbl.gov/`, 2011.

[23] Bern, M., Gilbert, J. R., Hendrickson, B., Nguyen, N., and Toledo, S. Support-graph Preconditioners. Tech. rep., SIAM Journal on Matrix Analysis and Applications, 2006.

[24] Bockelmann, H. *High Performance Computing Based Methods for Simulation and Optimisation of Flow Problems.* PhD thesis, Karlsruhe Institute of Technology, 2010.

[25] Boman, E. G., Chen, D., Hendrickson, B., and Toledo, S. Maximum-Weight-Basis Preconditioners. *Applications 29* (2002), 695–721.

[26] Boman, E. G., Guattery, S., and Hendrickson, B. Optimal Embeddings and Eigenvalues in Support Theory.

[27] Boman, E. G., and Hendrickson, B. Support Theory for Preconditioning. *SIAM Journal on Matrix Analysis and Applications 25* (2001), 694–717.

[28] Boman, E. G., Hendrickson, B., and Vavasis, S. Solving Elliptic Finite Element Systems in Near-linear Time with Support Preconditioners. *CoRR 407022* (2004).

[29] Braess, D. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*, 2 ed. Cambridge Univ. Press, Cambridge, 2001.

[30] Brenner, S. C., and Scott, L. R. *The Mathematical Theory of Finite Element Methods*, 2 ed. Texts in Applied Mathematics ; 15. Springer, New York, 2002.

[31] Butcher, J. C. *Numerical Methods for Ordinary Differential Equations*, 2 ed. Wiley, Chichester, 2008.

[32] Börm, S., Grasedyck, L., and Hackbusch, W. An Introduction to Hierarchical Matrices. *Math. Bohem 127* (2003), 229–241.

[33] Cai, X., and Sarkis, M. A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems. *SIAM J. Sci. Comput 21* (1999), 792–797.

[34] Chen, D. Analysis, Implementation, and Evaluation of Vaidya's Preconditioners, 2001.

[35] Chen, K. *Matrix Preconditioning Techniques and Applications*, 1 ed. Cambridge Monographs on Applied and Computational Mathematics ; 19. Cambridge University Press, Cambridge, 2005.

[36] Chow, E., and Saad, Y. Parallel Approximate Inverse Preconditioners. In *Proceedings of the Eight SIAM Conference on Parallel Processing for Scientific Computing, SIAM* (1997), pp. 14–17.

[37] Ciarlet, P. G. *The Finite Element Method for Elliptic Problems.* Classics in Applied Mathematics; 40. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2002.

[38] Cuthill, E., and McKee, J. Reducing the Bandwidth of Sparse Symmetric Matrices. In *ACM '69: Proceedings of the 1969 24th National Conference* (New York, NY, USA, 1969), ACM, pp. 157–172.

[39] Datta, K. *Auto-tuning Stencil Codes for Cache-based Multicore Platforms.* PhD thesis, University of California, Berkeley, 12 2009.

[40] Demkowicz, L. F. Computing with hp-adaptive Finite Elements, 2006.

[41] Demmel, J., Dongarra, J., Ruhe, A., and van der Vorst, H. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[42] Demmel, J. W. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, PA, 1997.

[43] Ern, A., and Guermond, J. *Theory and Practice of Finite Elements.* Applied Mathematical Sciences ; 159. Springer, New York, 2004.

[44] Fraunhofer-Gesellschaft. LAMA - Library for Accelerated Mathematical Applications `http://www.libama.org/`.

[45] Gander, W., G, W., and Golub, G. H. Cyclic Reduction - History and Applications.

[46] Glowinski, R. *Numerical Methods for Nonlinear Variational Problems.* Springer series in computational physics. Springer, New York, 1984.

[47] GNU MP. The GNU Multiple Precision Arithmetic Library `http://gmplib.org/`, 2011.

[48] Göddeke, D. *Fast and Accurate Finite Element Multigrid Solvers for PDE Simulations on GPU Clusters.* PhD thesis, Technische Universität Dortmund, Fakultät für Mathematik, 2010.

[49] Goldberg, D. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv. 23* (March 1991), 5–48.

[50] Golub, G., and Van Loan, C. *Matrix Computations*, 3 ed. Johns Hopkins Series in the Mathematical Sciences. Johns Hopkins Univ. Pr., Baltimore, Md., 1997.

[51] Griebel, M., Dornseifer, T., and Neunhoeffer, T. *Numerische Simulation in der Strömungsmechanik.* Vieweg, 1995.

[52] Grossmann, C., and Roos, H. *Numerical Treatment of Partial Differential Equations.* Universitext. Springer, Berlin, 2007.

[53] Grote, M. J., and Huckle, T. Parallel Preconditioning with Sparse Approximate Inverses. *SIAM J. Sci. Comput 18* (1996), 838–853.

[54] Hackbusch, W. *Elliptic Differential Equations: Theory and Numerical Treatment.* Springer Series in Computational Mathematics ; 18. Springer, Berlin, 1992.

[55] Hackbusch, W. *Hierarchische Matrizen: Algorithmen und Analysis.* Springer, 2009.

[56] Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach*, 4 ed. Elsevier, Amsterdam, 2007.

[57] Heuveline, V., Lukarski, D., Subramanian, C., and Weiss, J. P. A Multiplatform Linear Algebra Toolbox for Finite Element Solvers on Heterogeneous Clusters. In *PPAAC'10, IEEE Cluster 2010 Workshops* (2010).

[58] Heuveline, V., Lukarski, D., Subramanian, C., and Weiss, J.-P. Parallel preconditioning and modular finite element solvers on hybrid CPU-GPU systems. In *Proceedings of ParEng 2011* (2011).

[59] Heuveline, V., Lukarski, D., Trost, N., and Weiss, J.-P. Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs `http://www.emcl.kit.edu/preprints/emcl-preprint-2011-09.pdf`. EMCL Preprint Series, 2011.

[60] Heuveline, V., Lukarski, D., and Weiss, J.-P. Performance of a Stream Processing Model on the Cell BE NUMA Architecture Applied to a 3D Conjugate Gradient Poisson Solver. *International Journal of Computational Science 3(5)* (2009), 473–490.

[61] Heuveline, V., Lukarski, D., and Weiss, J.-P. Scalable Multi-coloring Preconditioning for Multi-core CPUs and GPUs. In *UCHPC'10, Euro-Par 2010 Parallel Processing Workshops* (Heidelberg, 2010), vol. 6586, Springer, LNCS, pp. 389–397.

[62] Heuveline, V., Lukarski, D., and Weiss, J.-P. Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method `http://www.emcl.kit.edu/preprints/emcl-preprint-2011-08.pdf`. EMCL Preprint Series, 2011.

[63] Heuveline, V., and Weiss, J.-P. Lattice Boltzmann Methods on the ClearSpeed Advance Accelerator Board. *Eur. Phys. J. Special Topics 171* (2009), 31–36.

[64] HiFlow^3. Parallel Finite Element Software `http://www.hiflow3.org/`.

[65] Hysom, D., and Pothen, A. Level-based Incomplete LU Factorization: Graph Model and Algorithms. Preprint UCRL-RC-150789.

[66] IBM. Programmer's Guide - SDK for Multicore Acceleration Version 3.0, 2007.

[67] IBM, 2008. PowerXCell 8i Processor http://www-03.ibm.com/technology/cell/pdf/PowerXCellPB7May2008pub.pdf.

[68] IEEE. The IEEE and The Open Group. The Open Group Base Specifications Issue 6, 2004.

[69] IEEE. Standard for Floating-point Arithmetic. *IEEE Std 754-2008* (29 2008), 1 –58.

[70] Intel. Cilk http://software.intel.com/en-us/articles/intel-cilk-plus-software-development-kit/, 2011.

[71] Intel. Dual Core / Yonah http://ark.intel.com/products/codename/2673/Yonah, 2011.

[72] Intel. Many Integrated Core Architecture http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html, 2011.

[73] Intel. Math Kernel Library (MKL) http://software.intel.com/en-us/articles/intel-mkl/, 2011.

[74] Intel. Single-Chip Cloud Computer http://techresearch.intel.com/ProjectDetails.aspx?Id=1, 2011.

[75] Intel. TBB http://threadingbuildingblocks.org/, 2011.

[76] Johnson, C. *Numerical Solution of Partial Differential Equations by the Finite Element Method.* Cambridge University Press, Cambridge, England, 1987.

[77] Kahan, W., 1996. Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-point Arithmetic, http://http.cs.berkeley.edu/ wkahan/ieee754status/ieee754.ps.

[78] Khronos Group. OpenCL http://www.khronos.org/opencl/, 2011.

[79] Kolotilina, L. Y., and Yeremin, A. Y. Factorized Sparse Approximate Inverse Preconditionings I: Theory. *SIAM J. Matrix Anal. Appl. 14* (January 1993), 45–58.

[80] Kwon, Y., and Bang, H. *The Finite Element Method Using MATLAB*, 2 ed. CRC Mechanical Engineering Series. CRC Press, Boca Raton, Fla., 2000.

[81] Liu, J. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Anal. Appl. 11* (January 1990), 134–172.

[82] Matrix Market. bcsstk06 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc1/bcsstk06.html, 2011.

[83] Matrix Market. bcsstk13 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc1/bcsstk13.html, 2011.

[84] Matrix Market. bcsstk14 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc2/bcsstk14.html, 2011.

[85] Matrix Market. bcsstk15 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc2/bcsstk15.html, 2011.

[86] Matrix Market. gr_30_30 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/laplace/gr_30_30.html, 2011.

[87] Matrix Market. nos1 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos1.html, 2011.

[88] Matrix Market. nos2 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos2.html, 2011.

[89] Matrix Market. nos3 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos3.html, 2011.

[90] Matrix Market. nos4 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos4.html, 2011.

[91] Matrix Market. nos5 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos5.html, 2011.

[92] Matrix Market. nos6 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos6.html, 2011.

[93] Matrix Market. nos7 http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos7.html, 2011.

[94] Matrix Market. s1rmt3m1 http://math.nist.gov/MatrixMarket/data/misc/cylshell/s1rmt3m1.html, 2011.

[95] Matrix Market. s3dkq4m2 http://math.nist.gov/MatrixMarket/data/misc/cylshell/s3dkq4m2.html, 2011.

[96] McCool, M. D. Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE 96*, 5 (2008), 816–831.

[97] Morton, K. W., and Mayers, D. F. *Numerical Solution of Partial Differential Equations: An Introduction*, 2 ed. Cambridge University Press, Cambridge, 2005.

[98] Naumov, M. Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. Tech. Rep. NVIDIA NVR-2011-001, NVIDIA, 2011.

[99] Netlib. http://www.netlib.org/, 2011.

[100] NVIDIA. CUDA http://developer.nvidia.com/what-cuda, 2011.

[101] NVIDIA. CUDA http://www.nvidia.com/content/cudazone/cuda_sdk/Linear_Algebra.html, 2011.

[102] NVIDIA. CUSPARSE http://developer.nvidia.com/cusparse, 2011.

[103] NVIDIA. GPU Computing Devices http://www.nvidia.com/object/personal-supercomputing.html, 2011.

[104] NVIDIA. GTX 580 http://www.nvidia.com/object/product-geforce-gtx-580-us.html, 2011.

[105] NVIDIA. HPC Computing with Tesla http://www.nvidia.com/object/tesla_computing_solutions.html, 2011.

[106] OpenMP. http://openmp.org, 2011.

[107] Press, W. H. *Numerical Recipes in C: The Art of Scientific Computing*, 2 ed. Cambridge Univ. Press, Cambridge, 1996.

[108] Przemieniecki, J. S. *Theory of Matrix Structural Analysis.* McGraw-Hill, New York, 1968.

[109] Quarteroni, A., and Valli, A. *Domain Decomposition Methods for Partial Differential Equations*, 1 ed. Numerical Mathematics and Scientific Computation. Clarendon Press, Oxford, 1999.

[110] Rocker, B. H. K. *Hardware-aware Solvers for Large, Sparse Linear Systems: Multi-precision and Parallel Approaches.* PhD thesis, Karlsruhe Institute of Technology, 2011.

[111] Roos, H., Stynes, M., and Tobiska, L. *Numerical Methods for Singularly Perturbed Differential Equations: Convection-diffusion and Flow Problems.* Springer Series in Computational Mathematics ; 24. Springer, Berlin, 1996.

[112] Rupp, K. ViennaCL `http://viennacl.sourceforge.net/`, 2011.

[113] Saad, Y. A Flexible Inner-outer Preconditioned GMRES Algorithm, 1993.

[114] Saad, Y. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[115] Saad, Y. Presentation Slides `http://www-users.cs.umn.edu/~saad/PDF/Nancy_06_09_2010.pdf`, 2010.

[116] Scott, J., and Tuma, M. The Importance of Structure in Incomplete Factorization Preconditioners. *BIT 51*, 2 (2011), 385–404.

[117] Shapira, Y. *Matrix-based Multigrid. Theory and Applications (Numerical Methods and Algorithms).* Springer, 2003.

[118] Shapira, Y. *Solving PDEs in C++: Numerical Methods in a Unified Object-oriented Approach.* Computational Science and Engineering. Society for Industrial and Applied Mathematics, Philadelphia, 2006.

[119] Sleijpen, G., and Fokkema, D. BICGSTAB(l) For Linear Equations Involving Unsymmetric Matrices With Complex Spectrum, 1993.

[120] Smith, B., Bjørstad, P., and Gropp, W. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge Univ. Press, Cambridge, 1996.

[121] Sokolnikoff, I. ; Redheffer, R. *Mathematics of Physics and Modern Engineering.* MacGraw-Hill, New York, 1958.

[122] Stoer, J., and Bulirsch, R. *Numerische Mathematik*, 4 ed., vol. 2 of *Springer-Lehrbuch.* Springer, Berlin, 2000.

[123] Toselli, A., and Widlund, O. *Domain Decomposition Methods - Algorithms and Theory.* Springer Series in Computational Mathematics ; 34. Springer, Berlin, 2005.

[124] Trottenberg, U., Oosterlee, C., and Schüller, A. *Multigrid.* Academic Press, Amsterdam, 2003.

[125] University of Florida Sparse Matrix Collection. ecology2 `http://www.cise.ufl.edu/research/sparse/matrices/McRae/ecology2.html`, 2011.

[126] University of Florida Sparse Matrix Collection. G3_circuit http://www.cise.ufl.edu/research/sparse/matrices/AMD/G3_circuit.html, 2011.

[127] University of Florida Sparse Matrix Collection. mesh3e1 http://www.cise.ufl.edu/research/sparse/matrices/Pothen/mesh3e1.html, 2011.

[128] University of Florida Sparse Matrix Collection. mhd3200b http://www.cise.ufl.edu/research/sparse/matrices/Bai/mhd3200b.html, 2011.

[129] University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/, 2011.

[130] van der Vorst, H. A. BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput. 13* (March 1992), 631–644.

[131] Vandebril, R., Barel, M. V., and Mastronardi, N. A Note on the Representation and Definition of Semiseparable Matrices, 2003.

[132] Vuduc, R. W. *Automatic Performance Tuning of Sparse Matrix Kernels.* PhD thesis, University of California, Berkeley, CA, USA, January 2004.

[133] Young, D. M. *Iterative Solution of Large Linear Systems.* Computer Science and Applied Mathematics. Acad. Press, New York, 1971.

[134] Yserentant, H., Vom, H., and Andre, V. D. K. On the Multi-level Splitting of Finite Element Spaces. *Numer. Math 49* (1986), 379–412.