Karlsruhe Institute of Technology

Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method

Vincent Heuveline
Dimitar Lukarski
Jan-Philipp Weiss

No. 2011-08

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)

$\frac{\partial u}{\partial t} - \nu \Delta u$
$+ (u \cdot \nabla) u + \frac{1}{\rho} \nabla p$
$\frac{\partial u}{\partial t}$

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
No. 2011-08

Impressum

Karlsruhe Institute of Technology (KIT)
Engineering Mathematics and Computing Lab (EMCL)

Fritz-Erler-Str. 23, building 01.86
76133 Karlsruhe
Germany

KIT – University of the State of Baden Wuerttemberg and
National Laboratory of the Helmholtz Association

Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method

Vincent Heuveline[1], Dimitar Lukarski[1,2,*] and Jan-Philipp Weiss[1,2]

[1] Engineering Mathematics and Computing Lab (EMCL)
[2] SRG New Frontiers in High Performance Computing
Karlsruhe Institute of Technology, Germany
{vincent.heuveline, dimitar.lukarski, jan-philipp.weiss}@kit.edu

Abstract. Application demands and grand challenges in numerical simulation require for both highly capable computing platforms and efficient numerical solution schemes. Power constraints and further miniaturization of modern and future hardware give way for multi- and manycore processors with increasing fine-grained parallelism and deeply nested hierarchical memory systems – as already exemplified by recent graphics processing units. Accordingly, numerical schemes need to be adapted and re-engineered in order to deliver scalable solutions across diverse processor configurations. Portability of parallel software solutions across emerging hardware platforms is another challenge.

This work investigates multi-coloring and re-ordering schemes for block Gauß-Seidel methods and, in particular, for incomplete LU factorizations with and without fill-ins. We consider two matrix re-ordering schemes that deliver flexible and efficient parallel preconditioners. The general idea is to generate block decompositions of the system matrix such that the diagonal blocks are diagonal itself. In such a way, parallelism can be exploited on the block-level in a scalable manner. Our goal is to provide widely applicable, out-of-the-box preconditioners that can be used in the context of finite element solvers.

We propose a new method for anticipating the fill-in pattern of ILU(p) schemes which we call the *power(q)-pattern method*. This method is based on an incomplete factorization of the system matrix A subject to a predetermined pattern given by the matrix power $|A|^{p+1}$ and its associated multi-coloring permutation π. We prove that the obtained sparsity pattern is a superset of our modified ILU(p) factorization applied to $\pi A \pi^{-1}$. As a result, this modified ILU(p) applied to multi-colored system matrix has no fill-ins in its diagonal blocks. This leads to an inherently parallel execution of triangular ILU(p) sweeps.

In addition, we describe the integration of the preconditioners into the HiFlow^3 open-source finite element package that provides a portable software solution across diverse hardware platforms. On this basis, we conduct performance analysis across a variety of test problems on multi-core CPUs and GPUs that proves efficiency, scalability and flexibility of our approach. Our preconditioners achieve a solver acceleration by a factor of up to 1.5, 8 and 85 for three different test problems. The GPU versions of the preconditioned solver are by a factor of up to 4 faster than an OpenMP parallel version on eight cores.

Keywords: Parallel preconditioners, fine-grained parallelism, multi-coloring, ILU with fill-ins, power(q)-pattern method, multi-core CPUs, GPU

[*] Corresponding author: dimitar.lukarski@kit.edu

1 Introduction

Numerical simulation and its huge computational demands require a close coupling between efficient mathematical methods and their hardware-aware implementation on emerging and highly parallel computing platforms. The paradigm shift towards manycore parallelism not only offers a high potential of computing capabilities but also comes up with urgent challenges in designing scalable, portable, flexible and numerical software solutions. The latter point is closely related to adaptation, variation, and re-structuring of algorithms and numerical schemes in order to be compliant with coarse- and fine-grained parallelism, hierarchical memory subsystems, heterogeneous platforms, and communication bottlenecks. Numerical codes should not only be efficient and robust, but also future-proof with respect to the current dynamic landscape of hardware platforms and parallel programming environments. Locality of computations and strategies for avoiding communication are the major building blocks for platform-optimized implementations. On the mathematical side, preconditioning techniques are a vital building block for linear system solvers for sparse problems arising e.g. finite element methods (FEM) or related techniques for the solution of partial differential equations (PDEs). The motivation of our work is to address highly complex problems occurring in areas such as meteorology, medical engineering and energy research, with the goal of using in an optimal way the existing multi-core CPU and GPU technology. These problems have in common that the resulting linear systems which need to be solved are usually highly coupled, very large and badly conditioned. With the claim of solving such problems with strong impact on science and society, the proposed preconditioning techniques should allow to push further beyond the capabilities of the modern computing platforms.

A classical choice for iterative solvers are Krylov subspace methods like *conjugate gradient* (CG) for symmetric and positive definite systems, the *generalized minimal residual method* (GMRES), or the *biconjugate gradient stabilized method* (BiCGstab) [31]. In all cases, the number of iterations depends on the condition number and grows polynomially in the problem size. While direct solvers offer very fast solutions for moderate problem sizes their applicability is limited for very large problems and their efficient parallelization is complex or limited. Krylow space-based iterative solvers can only reach acceptable performance if paired with efficient preconditioning schemes. Good preconditioners should fulfill several properties. First, they should mitigate the costs in terms of necessary iterations by restructuring the problem matrix and affecting its spectrum and condition number. Second, since in each iteration step an additional linear system has to be solved, the additional effort should not outweigh achieved benefits. Third – and this point is becoming much more important due to the development towards manycore computing platforms – the preconditioner has to comprise a high degree of parallelism. Fourth, the preconditioner should be applicable to a large class of problems where only minimal additional information is available on specific matrix properties. The latter point is particularly important for the inclusion of preconditioners into widely applicable solver suites. As experience shows, parallelism for preconditioners based on reduced couplings comes at the expense of reduced preconditioning efficiency. So preconditioning also means to find a trade-off between several aspects.

The main goal of our work is to provide parallel preconditioners with a speedup in two dimensions: first, there should be a parallel speedup and scalability properties when running on additional cores. Second, the preconditioner should yield reduced total solver time compared to the unpreconditioned parallel solver. By means of the multi-platform approach in the context of the HiFlow^3

project [21], the solver and preconditioner suite is maximally flexible, integrates with the full FEM solver stack, and runs on diverse hardware platforms. Furthermore, the preconditioner suite should be an out-of-the-box approach applicable to large classes of problems and matrices and should not be restricted to specific cases. The considered techniques in this work only analyze the matrix structure and do not rely on additional information (e.g. the matrix spectrum, underlying discretization or grid). As flexibility of solution techniques and software is a decisive factor in the current computing landscape, we have incorporated our preconditioners into the multi-platform parallel finite element software package HiFlow^3, that is portable across diverse computing platforms. HiFlow^3 tackles productivity and performance issues by its conceptual approach. Efficient numerical solvers are built on the basis of unified interfaces to different hardware platforms and parallel programming approaches in order to obtain modular and portable software solutions on emerging systems with heterogeneous configuration. With the concept of object-orientation in C++, the HiFlow^3 finite element software provides a flexible, generic and modular framework for building efficient parallel solvers and preconditioners for partial differential equations of various types.

In this work we propose a new preconditioning approach which relies on an enhanced multi-colored ILU(p) factorization with predetermined fill-ins. This numerical method has been successfully developed and tested in the framework of the freely available software package HiFlow^3. It should be emphasized that the realization of the proposed approach is a key and non-trivial in order to be able to objectively evaluate the quality of the method. This is especially due to the fact that the derivation of the this technique involves arguments related to numerical analysis, algorithmic and computer architecture, which need to be taken into account in a holistic way.

Our main intention is to construct preconditioners with a high degree of parallelism. Structure and organization of the HiFlow^3 software into several modules and communication layers with unified interfaces for the programmer allows to write a single code base that can be run across a wide range of parallel platforms including multi-core CPUs, graphics processing units (GPUs), and OpenCL-capable accelerators. Since GPU code should be scalable to thousands of threads, our idea is to identify parallelism on the level of blocks within block-decompositions and not only on the level of non-scalable parallel execution of blocks. We consider preconditioners in block form based on additive matrix splittings, like e.g. Gauß-Seidel and SOR, and multiplicative decompositions like incomplete LU (ILU). All our considered preconditioners are based on the blockwise decomposition into small sub-matrices. In both scenarios, typically a large amount of forward and backward sweeps in triangular solvers need to be performed. In order to harness parallelism within each block of the decomposition we use matrix re-ordering techniques like multi-coloring and level scheduling. For splitting-based methods (Gauß-Seidel, SOR) and ILU(0) without fill-ins the original matrix occupancy pattern of additive or multiplicative decompositions is preserved in the sense, that only subsets of the original occupancy pattern are populated and no additional matrix elements are inserted. Before solving the preconditioned system, the matrix is reorganized such that diagonal blocks in the block decomposition are diagonal itself. Then, inversion of the diagonal blocks is just an easy vector operation [22]. Furthermore, we allow fill-ins (i.e. additional matrix elements) in the ILU(p) method for achieving a higher level of coupling with increased efficiency. Here, we consider two algorithms for parallelism: *level-scheduling* method [31] and our *power(q)-pattern method* combined with multi-coloring. The level scheduling method [31] is used as a postprocessing method following the factorization. The level of parallelism for the elimination processes in the forward and backward sweeps is determined and utilized. However, this method produces very small blocks for many problem classes, i.e. the degree of parallelism is low.

In this work, we present a new parallel algorithm for performing ILU(p) with fill-ins by means of the *power(q)-pattern method*. The major advantage is that multi-coloring can be applied to this new structure before performing the ILU(p) with additional fill-ins. The diagonal structure of the block diagonals is then preserved – fill-ins only occur outside the blocks on the diagonal. By precomputing a superset of the data distribution pattern we eliminate costly insertion of new matrix elements into dynamic data structures. We restrict our work to node-level preconditioners either executed on shared memory based multi-core system with several sockets and to GPU-enhanced systems with highly competitive raw performance. Other accelerators can easily be included by means of our OpenCL backends. We are currently working on hybrid solvers running on several node-level devices in parallel.

This paper is structured as follows. Section 2 gives a short overview of related work on parallel and cross-platform software implementations of sparse solvers and preconditioners. In Section 3 we describe the basic methodology of the block-level parallel preconditioners. We give an outline of the level scheduling and multi-coloring methods. Moreover, we propose our power(q)-pattern enhanced multi-colored ILU(p,q) scheme. We prove that for our modified incomplete factorization the sparsity pattern of the multi-colored matrix power $|A|^{p+1}$ is a superset for the sparsity pattern of the permuted system matrix A. In such a way, we obtain full control on the fill-in elements that do not disturb the diagonal structure of the re-arranged diagonal blocks. This observation is the basis for parallelism and efficiency of our presented preconditioner. Section 4 details the concept of the *lmpLAtoolbox* that provides a flexible and user-friendly framework for building widely applicable and portable numerical software. The impact of re-ordering schemes and our preconditioners on the sparsity patterns of small test matrices is investigated in Section 5. In Section 6 we present a detailed performance analysis on an 8-core shared memory machine and a single GPU platform for realistic scenarios – proving viability and benefit of our solution. In combination with HiFlow^3's cross-platform portable concept we provide flexible and scalable iterative solvers combined with efficient parallel preconditioners based on splitting methods and ILU. Some remarks on our intended future work are summarized in Section 7. Our work concludes in the final Section 8.

2 Related Work on Parallel Implementations of Sparse Solvers and Preconditioners

There exist a whole bunch of parallel solvers and solution libraries such as PETSc [4], Trilinos [20], MKL [24], or CUSP [8]. Most of them are limited to specific platforms (cluster of CPUs or single GPU) and are hence not fully portable, or they are limited in their functionality and only simple preconditioners are implemented. Many algorithms are still designed for fat cores where data is partitioned into large blocks that are processed sequentially on the cores with minimal communication across the cores. This concept does not apply to fine-grained parallelism as required on GPUs

The CUSP library [8] provides a high-level interface for functions for sparse linear algebra and graph computations on CUDA-enabled GPUs. Several preconditioners are under development – with currently only diagonal scaling by

Jacobi implemented. PETSc-dev [4] has some support for running parts of computations on NVIDIA GPUs on top of MPI by means of the VECCUSP and MATCUSP classes, but preconditioners are limited to the simple block-Jacobi case. Preconditioners in Trilinos comprise ILU-type preconditioners (Ifpack), multi-level (ML) preconditioners and block-type preconditioners (Meros, Teko) – with all of them restricted to CPU platforms. ViennaCL [30] is a suite of solvers and software package based on OpenCL implementations. It also contains some preconditioners and runs on single GPUs as well as on multi-core CPUs but the ILU preconditioners are only for the CPU backend.

Highly parallel preconditioners can be obtained by directly approximating the inverse matrix. One can obtain the approximate matrix by the matrix-valued Chebyshev polynomials but in this case an approximation of spectrum is needed, a GPU implementation is presented in [2]. The Factorized Sparse Approximate Inverse (FSAI) [26] technique is a promising approach to build generic and parallel preconditioners but to our knowledge so far there is no GPU implementation of this algorithm. There are few works on multi-grid methods which are used as preconditioner schemes. In [17] the multi-grid method is used as highly parallel preconditioner of the outer loop iterative method. However, this approach is limited to topological structured meshes and does not allow arbitrary order of finite elements.

In our previous work we have presented a performance evaluation of the block Jacobi and multi-colored symmetric Gauss-Seidel preconditioner on multi-core CPU and GPU configurations [22]. A performance investigation of the preconditioned solution scheme for a convection-diffusion problem solved by Q1 and Q2 elements in two and three dimensions is presented in [32] where Gauss-Seidel and ILU preconditioners without fill-ins are considered. A multi-level linear algebra library design and a performance analysis on hybrid-parallel linear solver on a GPU cluster with eight nodes and sixteen GPUs can be found in in [23]. These results underline good scalability properties of our solver libraries across different devices.

3 Parallel Preconditioners

Preconditioners are used in the context of iterative solvers for decreasing the number of necessary iterations for reaching a prescribed error tolerance. This kind of techniques can be successfully used to affect the condition number of the system matrix. There are several classes of preconditioners used in numerical simulation based on sparse matrices.

Splitting-type preconditioners are based on additive splittings of the system matrix. Classical schemes are Jacobi, Gauß-Seidel, their block versions, and relaxed variants, e.g. SOR. Multiplicative factorizations like incomplete LU factorization maintain certain sparsity patterns or allow fill-ins into designated matrix elements increasing the number of non-zero elements. In the case of approximate inverse techniques the inverse of the system matrix is approximated on the basis of prescribed non-zero patterns. Some detailed descriptions on preconditioners can be found in the books [31, 16, 13]. Unfortunately, there is no fundamental theory on mathematical mechanisms like convergence or efficiency. There is only some insight with respect to special matrices related to some well studied equations on simple domains [3].

From a practical point of view it is important to decrease the total amount of time for the iterative solver and not only the number of iterations. Therefore, it is a key point to perform the preconditioning step as fast and as cheap as possible while maintaining the iteration reduction property. The LU-type preconditioners based on multiplicative factorization are very popular due to the fact that no information on matrix properties (e.g. eigenvalue spectrum) is needed. It is based on a Gauß elimination process. In the solution phase the forward and backward steps are typically processed in sequential order with linear complexity $O(N)$ or $O(nnz)$ (where $N \times N$ is the matrix dimension and nnz is the number of non-zero matrix elements). But with increasing need for parallelism the main question arising is how to efficiently solve the forward and backward steps in parallel.

Our goal is to provide flexible, scalable and generic fine-grained parallel techniques for performing the forward and backward sweeps in parallel on any parallel computing platform by means of incomplete LU-factorization with or without fill-in elements.

3.1 Properties of sparse matrices arising from finite element methods

Finite elements methods (FEM) are a common technique for solving linear and non-linear partial differential equations (PDEs), see e.g. [14, 10]. In our work we are focusing on Galerkin-FEM for stationary PDEs with identical ansatz and test spaces on conforming and non-conforming (hanging nodes allowed with respect to h- or p-non-conformity) polyhedral meshes with arbitrary polynomial order of finite element ansatz functions. Meshes may be refined locally by h-, p, hp- or r-refinement. By assembling the local element contributions the global system matrix A (stiffness matrix plus mass matrix plus non-symmetric contributions) is obtained – representing a linear system of equations. Boundary conditions include Dirichlet, Neumann or Robin type. In the non-linear case typically a linearization is paired with a Newton-type approach. Since localized finite elements with finite support result in near neighbor interaction, the system matrix A is typically sparse. We denote the non-zero matrix pattern (sparsity or occupancy pattern) of an N-by-N matrix $A = (a_{ij})_{i,j=1,\ldots,N}$ by

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

If the square matrix A is symmetric or does not depend on the underlying differential operators (discretized by means of a variational formulation with bilinear forms in the FE-context) where the non-symmetric part is often related to a convection or transport term. For coercive bilinear forms the diagonal elements of the system matrix are positive [11, 10, 18]. In the sequel we assume that all diagonal elements are non-zero.

3.2 Blockwise preconditioners

The basic idea of the considered preconditioners is to decompose the matrix into an agglomerate of smaller blocks with a lower bound in size. In our context, the level of parallelism is not determined by the number of blocks but by the number of elements per block. In the parallel preconditioner, we are iterating over the blocks on the diagonal in a sequential sense where each block or block row is processed in parallel. The number and size of the blocks in the decomposition can be given by user choice (like for block Jacobi methods) or can be 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. Matrix decompositions are either additive (splitting-type methods) or multiplicative (ILU-type methods). In the latter case, there is a preprocessing step for producing the matrix factorization. Moreover, matrix re-orderings are performed in the preprocessing phase as well.

The main idea of the matrix re-ordering schemes – i.e. permutations of the node numberings – is to preserve and exploit the sparse structure of the system matrix. In some cases, additional fill-ins are permitted and are a necessary building block for maintaining problem-inherent couplings. Special measures will be taken to anticipate the fill-in-pattern and to prevent fill-ins into the diagonal blocks. On modern hardware platforms performance for many kernels is mostly limited by bandwidth. Bandwidth-bound kernels should exploit as much locality as possible. And as all matrix re-ordering techniques may have a significant impact on locality of computations and cache performance, specific care has to be taken on this issue.

3.3 Preconditioners based on splitting methods

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

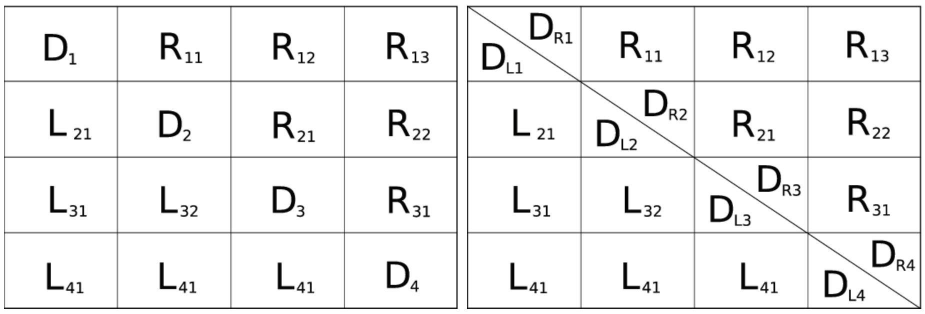

Fig. 1. Example of 4-by-4 block-decomposed matrix for $B = 4$; additive splitting for Gauß-Seidel-type (GS) methods (left) and multiplicative splitting for ILU-type methods (right)

In the block-Jacobi (BJ) case – the simplest version of a parallel preconditioner – block sizes can be chosen arbitrarily. Here, we take $M := D$ and solve the preconditioning equation $Mz = r$ by local inversion of $D_i z_i = r_i$, i.e. $z_i = D_i^{-1} r_i$ for block vectors r_i and z_i of length b_i. As a local solver (inversion of D_i) Gauß-Seidel (GS), LU-decomposition, or any direct method can be used. The degree of parallelism is given by the (artificial) number of blocks. Although arbitrarily parallel (at the expense of reduced inter-block couplings), mathematical efficiency of the block-Jacobi preconditioner is mostly disappointing.

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

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

$$y_i = D_i x_i \text{ for } i = 1, \ldots, B, \tag{2}$$

$$z_i = D_i^{-1}(y_i - \sum_{j=1}^{B-i} R_{ij}z_{i+j}) \text{ for } i = B, \ldots, 1, \tag{3}$$

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

The bracket expressions in the right hand sides of (1) and (3) now consist of $i-1$ and $B-i$ matrix-vector products with vector length b_j and b_{i+j}. In total B^2 sparse matrix-vector products and $2B$ sparse matrix inversions are necessary to compute (1)-(3). The degree of parallelism in each step is b_i for block row-wise execution (assuming parallel inversion of D_i). The value of b_i is N/B for uniform block size and is typically much larger than B. The major difficulty in computing (1)-(3) arises from parallel solution for the diagonal blocks D_i which are non-diagonal itself in general.

Other splitting type preconditioning schemes based on the original matrix structure are relaxed schemes like *successive overrelaxation* (SOR) and *symmetric SOR* (SSOR) [31, 3]. The following list summarizes various splitting-type decompositions based on additive decomposition $A = D + L + R$ (interpretation of block and non-block variants is straightforward):

$$\begin{aligned}
M_{\text{Jac}} &:= D, \\
M_{\text{GS}} &:= D + L, D + R \text{ (forward/backward version)}, \\
M_{\text{SGS}} &:= (D+L)D^{-1}(D+R), \\
M_{\text{SOR}} &:= \frac{1}{\omega}(D + \omega L), \frac{1}{\omega}(D + \omega R), \\
M_{\text{SSOR}} &:= \frac{1}{\omega(2-\omega)}(D+\omega L)D^{-1}(D+\omega R).
\end{aligned}$$

3.4 Preconditioners based on ILU decompositions

A similar idea applies to incomplete LU (ILU) decompositions. Here, the matrix is decomposed into a product $A = LU + R$ of a lower triangular matrix L and an upper triangular matrix U, and a remainder matrix R. Typically, diagonal entries of L are taken to be one 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 1 (right). In the sparse case, the sparsity pattern of A is preserved by ensuring $\mathcal{N}(L) \cup \mathcal{N}(U) \subseteq \mathcal{N}(A)$ (not equal since some elements might be deleted due to cancellation). Additional fill-in elements are put into the remainder matrix.

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

$$x_i = D_{Li}^{-1}(r_i - \sum_{j=1}^{i-1} L_{ij}x_j) \text{ for } i = 1, \ldots, B, \tag{4}$$

$$z_i = D_{Ri}^{-1}(x_i - \sum_{j=1}^{B-i} R_{ij}z_{i+j}) \text{ for } i = B, \ldots, 1. \tag{5}$$

Here again, each block row i, $i = 1, \dots, B$, has $i-1$ left blocks L_{ij}, $j = 1, ..., i-1$, and $B - i$ right blocks R_{ij}, $j = 1, \dots, B - i$. The diagonal blocks D_{Li} and D_{Ri}, $i = 1, \dots, B$, are lower and upper diagonal square matrices with size $b_i \times b_i$, but b_i may be different for all i. The vectors x_k and z_k, $k = 1, \dots, B$, are block vectors of length b_k. The bracket expressions in the right hand sides of (4) and (5) now consist of $i - 1$ and $B - i$ matrix-vector products with vector length b_j and b_{i+j}. In total $B^2 - B$ sparse matrix-vector products and $2B$ sparse matrix-inversions are necessary to compute (4) and (5). The degree of parallelism is b_i (assuming parallel inversion of D_{Li}, D_{Ri}). The major difficulty in computing (4) and (5) arises again from solving for the diagonal blocks D_{Li} and D_{Ri} which are non-diagonal itself in general.

Using data parallel BLAS 1 and BLAS 2 routines on the level of blocks per row, i.e. block-wise execution and not block row-wise execution, decreases the degree of parallelism by $1/B$ – resulting in a degree of parallelism of b_k/B, which is N/B^2 for equal block sizes. Usage of standard routines is a key point exploited by our software package which is described later in Section 4. As an alternative, problem-specific kernels could be used.

One of the drawbacks of the ILU decomposition is the possible breakdown of the procedure when pivoting is not applied. For some matrix types (e.g. diagonally dominant matrices) proper processing of the decomposition is ensured without pivoting. Pivoting is difficult for our proposed power(q)-pattern enhanced multi-colored ILU(p, q) scheme (see following sections). In this case a permutation based on the multi-coloring classification of the matrix could be performed. These variations are subject of current investigation.

3.5 Multi-coloring and parallel sparse triangular solvers

Inversion of the diagonal blocks D_i (and D_{Li}, D_{Ri} resp.) can be easily handled by applying multi-coloring as a preprocessing step. Triangular solvers are then reduced to 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 neighbor dependencies by introducing neighbor-ship classes (colors) such that for non-zero matrix elements $a_{ij} \in \mathcal{N}(A)$ with $A = (a_{ij})_{i,j=1,\dots,N}$ 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 1, see [31].

Algorithm 1 Multi-coloring

```
for $i = 1$ to $N$ do
    Set color($i$)=0
end for
for $i = 1$ to $N$ do
    Set color($i$)=min($k > 0 : k \neq$color($j$) for $j \in$ Adj($i$));
end for
```

Here, $\mathrm{Adj}(i) = \{j \neq i \,|\, a_{i,j} \neq 0\}$ are the adjacent nodes to node i. By renumbering the mesh nodes by colors the diagonal blocks D_i become diagonal itself. Then B is the number of colors, and b_k is the number of elements for color k. Inversion of the diagonal matrix then is only a component-wise scaling of the source vector. Due to the data parallelism of the associated matrix-vector and 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). The output of the multi-coloring algorithm is the color classification of the nodes – a vector containing the color ID of each node. Defining a 2-column vector with original index and color ID and sorting it by colors defines the requested permutation.

3.6 Parallel LU sweeps

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 symmetric permutation π that rearranges A in such a way that we obtain only diagonal elements in its diagonal blocks $\tilde{D}_{Li}$ and $\tilde{D}_{Ri}$ (cf. Figure 1 (right)) with $\pi(A) = \tilde{L}\tilde{U} + \tilde{R}$. This problem can be solved by using the multi-coloring Algorithm 1. Based on the index colors we can build π by re-ordering the nodes by groups of colors. Due to the fact that all the nodes from the same color have no adjacent nodes, the permuted matrix has only diagonal elements in its diagonal blocks. Furthermore, we exploit that splitting-type preconditioners and basic ILU(0) preconditioners preserve the matrix structure, i.e. diagonal blocks with only diagonal elements are preserved. Based on the multi-coloring permutation of the matrix we can perform all matrix inversions and matrix-vector multiplications on the block-level in fully parallel manner. Due to their diagonal structure the matrices D_i in (1) and (3) and D_{L_i}, D_{R_i} in (4) and (5) can be inverted easily by a simple vector operation.

3.7 Incomplete LU preconditioners

An important class of preconditioners are based on incomplete LU (ILU) factorization. In this context, we are looking for a lower-triangular matrix L and an upper-triangular matrix U with $A = LU + R$, where the remainder R satisfies some criteria and the preconditioner is chosen as $M := LU$. The ILU(0) decomposition does not allow any fill-in, i.e. $\mathcal{N}(L) \cup \mathcal{N}(U) \subseteq \mathcal{N}(A)$. In order to obtain the matrices L and U we perform Gaussian elimination. In the general case, in this process the sparse matrices L and U suffer from fill-in and have more non-zero elements than the input matrix A. In the sequel, we consider two algorithms for producing an appropriate sparse structure for the resulting lower and upper matrices.

3.8 ILU(0) with sparsity pattern based on the original matrix

Producing the ILU(0) factorization where $\mathcal{N}(L) \cup \mathcal{N}(U) \subseteq \mathcal{N}(A)$ can be done by means of Algorithm 2. For easy matrix inversion in (4) and (5) we need to obtain a block-decomposed matrix A where its diagonal blocks only have non-zero diagonal elements. Therefore, we perform a multi-coloring permutation on the original matrix A and after that we perform the ILU(0) factorization. The latter factorization preserves the arranged matrix structure. The loop order is arranged such that row-wise processing can be performed in the inner loop - favoring sparse data structures like CSR [5, 31].

3.9 ILU(p) with fill-in elements

The quality of the ILU factorization depends on the sparsity pattern of the resulting matrices L and U. Therefore, in order to increase the quality of the

Algorithm 2 Incomplete LU-factorization without fill-in elements - ILU(0)

```
for $i = 2$ to $N$ do
  for $k = 1$ to $i - 1$ and $(i, k) \in \mathcal{N}(A)$ do
    $a_{ik} = a_{ik}/a_{kk}$
    for $j = k + 1$ to $N$ and $(i, j) \in \mathcal{N}(A)$ do
      $a_{ij} = a_{ij} - a_{ik}a_{kj}$
    end for
  end for
end for
```

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 3. Details on this ILU(p) technique with level-p fill-ins can be found in [31, 13].

Algorithm 3 Incomplete LU-factorization with fill-in elements - ILU(p)

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

One of the main difficulties of the algorithm is to predict the new non-zero pattern of the resulting factorization matrices for $p > 0$. There should be no fill-in into the diagonal blocks. Without preliminary information on the distribution of the inserted elements the costs for allocating memory and updating the matrix by means of dynamical data structures can be significant. Unfortunately, there is no general answer how multi-coloring affects the locality structure of a specific matrix. In Section 5 we show the influence of multi-coloring and level scheduling on the sparsity patterns of two small test matrices.

3.10 Level-scheduling algorithm

As for the ILU(0) case, we want to perform the forward step (4) and the backward step (5) for the ILU(p) factorization with fill-ins in parallel. Since the sparsity pattern of L and U does not correspond to the sparsity pattern of A anymore due to the fill-ins, direct inversion cannot be applied. Moreover, multi-coloring on the level of the L and U matrices cannot be applied since this would destroy their upper and lower diagonal structure (no exchange of elements allowed over the diagonal). Therefore, we apply the idea of *level-scheduling.* Here, we need to sort the unknowns in a way that the i-th equation only depends on the previous $i - 1$ unknowns. This algorithm is called *topological sorting.* An algorithm with linear time complexity is the *level scheduling* method proposed in [31]. The level scheduling algorithm for a lower triangular matrix given in Algorithm 4 defines levels of depth for all rows $i = 1, \dots, N$ corresponding to a node i in the adjacency graph and to the variable i respectively. Then we can create a

Algorithm 4 Level scheduling algorithm

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

permutation matrix π that groups all the nodes with the same depth. The degree of parallelism is then 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. Slightly better results can be obtained if the ILU(p) factorization is preceded by a multi-coloring step, and level scheduling is applied afterwards. This additional step decreases the number of levels in comparison to the version without multi-coloring permutation. However, this improvement is not significant in many cases.

In our approach, the number of sparse matrix-vector multiplications 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 (e.g. for preventing function call overheads for small sub-matrices). Furthermore, the number of elements per level should not be too small.

3.11 The *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. In this subsection we derive an upper bound for the non-zero pattern of a modified matrix factorization.

The non-zero pattern of the ILU(p) factorization matrix looks very similar to the matrix-matrix multiplication pattern. The sparsity pattern after the factorization shows that the non-zero pattern of ILU(p) grows like $|A|^{p+1}$. Inspired by this fact we can restrict the non-zero pattern of the factorization by determining the pattern of $|A|^{p+1}$ in order to avoid dynamic memory allocation. In addition, a multi-coloring step allows rearrangement of the diagonal blocks on the basis of the pre-determined pattern. A modification of the original algorithm is presented in Algorithm 5.

This variation of the original ILU(p) scheme (cf. Algorithm 3) 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 full control over the sparsity patterns of the factor matrices L_p and U_p. More precisely we find:

Algorithm 5 Power(q)-pattern enhanced ILU(p,q) with $q = p + 1$

```
Determine sparsity pattern $\mathcal{N}(|A|^{p+1})$ of matrix power $|A|^{p+1}$
Set lev$(a_{ij}) = 0$ for all non-zero elements $a_{ij} \in \mathcal{N}(A)$, lev$(a_{ij}) = \infty$ otherwise
for $i = 2$ to $N$ do
  for $k = 1$ to $i - 1$ and $(i, k) \in \mathcal{N}(|A|^{p+1})$ with lev$(a_{ik}) \leq p$ do
    $a_{ik} = a_{ik}/a_{kk}$
    for $j = k + 1$ to $N$ and $(i, j) \in \mathcal{N}(|A|^{p+1})$ do
      $a_{ij} = a_{ij} - a_{ik}a_{kj}$
      lev$(a_{ij}) = \min(\text{lev}(a_{ij}), \text{lev}(a_{ik}) + \text{lev}(a_{kj}) + 1)$
    end for
  end for
  for $j = 2$ to $N$ do
    if lev$(a_{ij}) > p$ then
      $a_{ij} = 0$
    end if
  end for
end for
Delete all entries $a_{ij}$ where $a_{ij} = 0$ (compress the matrix due to possible erasement)
```

Proposition 1. *Let L_p and R_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 5. Then we have $\mathcal{N}(L_p) \cup \mathcal{N}(U_p) \subseteq \mathcal{N}(|A|^{p+1})$.*

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

There is no difference in the factorization results between Algorithm 5 and the original ILU(p) Algorithm 3 in the cases $p = 0$ and $p = 1$. For $p \geq 2$ the original algorithm might produce slightly larger non-zero patterns for general sparse matrices. However, for all of our studied cases 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 2. *Let $A = (a_{ij})_{i,j=1,...,N}$ be a matrix with all non-zero elements on its diagonal, i.e. $a_{ii} \neq 0$ 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 $\hat{A} := \pi|A|^q\pi^{-1}$ that is a block-decomposed matrix with only diagonal elements in its diagonal blocks. With $\tilde{A} := \pi|A|\pi^{-1}$ we find $\tilde{A}^l = \pi|A|^l\pi^{-1}$ for all l and $\tilde{A}^q = \hat{A}$. We also find $|\tilde{A}| = \tilde{A}$. If a positive matrix element b_{nm} of a non-negative matrix $B = (b_{ij})$ is given and the non-negative matrix $C = (c_{ij})$ has positive diagonal elements, i.e. $c_{kk} > 0$ for all k, then the corresponding element $(BC)_{nm}$ of the matrix product BC is positive due to $(BC)_{nm} = \sum_k b_{nk}c_{km} \geq b_{nm}c_{mm} > 0$. By this we conclude $\mathcal{N}(\tilde{A}) = \mathcal{N}(|\tilde{A}|) \subseteq \mathcal{N}(|\tilde{A}|^l) = \mathcal{N}(\tilde{A}^l) \subseteq \mathcal{N}(|\tilde{A}|^q)$ for all positive integers $l \leq q$. And so we find $\mathcal{N}(\pi A\pi^{-1}) \subseteq \mathcal{N}(\pi|A|^l\pi^{-1}) \subseteq \mathcal{N}(\pi|A|^q\pi^{-1})$ for every positive integer $l \leq q$. Hence, $\pi A\pi^{-1}$ and $\pi|A|^l\pi^{-1}$, $1 \leq l \leq q$, are block-decomposed matrices where the diagonal blocks only have diagonal elements.

Now, we combine Proposition 1 and Proposition 2 to formulate Proposition 3:

Proposition 3. *Let $A = (a_{ij})_{i,j=1,\dots,N}$ be a matrix with all non-zero elements on its diagonal, i.e. $a_{ii} \neq 0$ for $i = 1,\dots,N$. Let π denote the multi-coloring permutation based on the matrix $|A|^{p+1}$ and $A_\pi := \pi A \pi^{-1}$ is 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 5 applied to A_π is producing two block-decomposed factor matrices L_p and U_p where the diagonal blocks only have diagonal elements. Fill-ins only occur outside the diagonal blocks.*

Proof. By Proposition 2 the matrix A_π is a block-decomposed matrix with only diagonal elements in its diagonal blocks. By applying Algorithm 5 to A_π we obtain matrix factors L_p and U_p with $\mathcal{N}(L_p) \cup \mathcal{N}(U_p) \subseteq \mathcal{N}(|A_\pi|^{p+1})$ due to Proposition 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 results in the following Algorithm 6, the *power(q)-pattern enhanced multi-colored ILU(p,q) method*. In the general case q will be taken as $q = p + 1$. Later, we will consider also the case $q \leq p$.

Algorithm 6 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 $\pi$
    – 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 5) to $A_\pi$
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(4) and (5)
    – use given number of colors $B$ and local block sizes $b_i$
```

This algorithm produces a block-decomposed system that reshapes the problem for parallel execution. 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 that obtained by the level-scheduling algorithm.

The proposed power(q)-pattern enhanced multi-colored ILU scheme corresponding to Algorithm 5 and Algorithm 6, denoted by ILU(p,q) in the following, is faster than the original method given in Algorithm 3 (denoted by ILU(p)). The main difference is that we have an upper bound for the sparsity pattern and elements where fill-ins happen are known in advance. The computation based on the predetermined sparsity pattern of $|A|^{p+1}$ is faster due to several aspects. First, we do not have to allocate large chunks of the data and compress the matrix afterwards but we only have to add some new elements at the end of the structure. Second, inner loops are much shorter and the length of the inner loops is known in advance. Consequently, there is no need for dynamic data structures and memory. Only at the end compression of the matrix is needed if new non-zero elements are obtained. And third and most important, the original ILU pattern (without multi-coloring and control of fill-ins) is not suited for parallel execution of the triangular sweeps.

The cost for building the sparsity pattern of $|A|^{p+1}$ is not negligible. However, the computation of this pattern is highly parallel and the complexity is much lower than the computation of $|A|^{p+1}$ itself. We do not have to perform to the whole inner loop of the matrix-matrix multiplication but only until the first non-zero entry appears. Furthermore, some optimization techniques like half looping for symmetric matrices can be applied.

Another difference in comparison to the original ILU algorithm is the permutation of the matrix before the factorization. This kind of permutation is based on the topology of the graph and not on the actual values. Additional permutation can be performed on the color classes in order to avoid zero pivot elements and breakdown of the factorization.

3.12 Increasing parallelism by drop-off techniques

We can increase the degree of parallelism by deleting selected elements of the obtained factorization matrices L and U. We describe two techniques: one for the level scheduling algorithm, and 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 since this comes at the expense of preconditioning efficiency and convergence of the iterative solver cannot be guaranteed.

Drop-off for level scheduling The elements for deletion 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 where all elements below this threshold are deleted. As an observation, this technique increases the level of parallelism only little but it decreases the quality of the preconditioner in many cases.

Drop-off for power(q)-pattern enhanced ILU(p, q) The number of colors in the power(q)-pattern enhanced 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, a fill-ins into the diagonal blocks are possible. These fill-in elements are selected for deletion. The effect of this 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. Note, that the number of sparse matrix-vector multiplications grows quadratically with respect to the number of colors obtained for the forward and backward step (of course the size of the matrices gets smaller). Therefore, the choice $q = p+1$ is no longer suitable for large p with respect to the time for the sweep steps. 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.

3.13 Other parallel preconditioners

In this subsection we provide a short overview of other classes of parallel preconditioners. Some of them are currently evaluated with respect to fine-grained parallelism on GPUs.

Approximate inverse The approximate inverse preconditioner tries to build a direct approximation to A^{-1}. But even if the matrix A is sparse there is no guarantee that the approximate inverse of A is sparse as well. For most problems, it is not feasible to store a dense matrix with the size of A. The main question is how to build an approximation with a certain sparsity pattern efficiently. A well studied approach is to approximate the matrix by Chebyshev matrix-valued polynomials. In this case, we obtain an approximation of the matrix where the sparsity pattern grows like A^k. But for building the matrix using Chebyshev polynomials we need to have information on the spectrum of the matrix – the largest and the smallest eigenvalue of the original problem. In most of the cases determination of this information has similar complexity as solving the original linear problem. But for a certain class of problems where the spectrum is known this preconditioning technique can be applied straightforwardly. Since sparse matrix-vector multiplication can be executed fully parallel this results in a highly parallel and efficient preconditioner. A GPU implementation with Chebyshev polynomials is presented in [2].

There are other approximate inverse techniques like SPAI [33] and FSAI [26]. Both of them do not require further information about the matrix like the eigenvalue distribution. The main drawback of the SPAI is the QR decomposition of a matrix in this algorithm which leads to a very long time for building the preconditioner. In contrast to that the FSAI algorithm does not have such factorization step and the coefficients for the sparsity pattern are computed by solving a large number of small linear systems.

Multigrid methods Multi-grid methods can be used as preconditioners for some outer loop iterative schemes. However, geometric multi-grid methods require information on the PDE, the underlying mesh, and the finite element space. Hence, this type of preconditioner cannot be used as an out-of-the-box solution for a given linear system. Some detailed information on geometric multi-grid methods on GPUs and other hardware can be found in [17].

On the other hand algebraic multi-grid (AMG) methods do not require any information on the underlying problem. Correspondingly, these schemes may be used as out-of-the-box preconditioners. But AMG has a complicated and time-consuming setup phase (mostly not parallel) which is based on heuristic arguments [19]. Nevertheless, this is a promising technique and it is addressed in the context of multi-core and many-core systems, e.g. in PyAMG [9].

Domain decomposition methods Domain decomposition methods can be used as parallel solvers and as preconditioning schemes. An example for that is the Restricted Additive Schwarz (RAS) method which is successfully mapped to a GPU system [15]. A method with coarse granularity is the Schur complement method. Successful mapping of Schur complement methods to GPUs needs to be proven.

Sparse direct methods Vaidya's preconditioner is an augmented maximum-weight-basis preconditioner that works by dropping non-zeros from the coefficient matrix and factorizing the remaining matrix, [12].

4 Local Multi-Platform LAtoolbox

Our preconditioners are implemented in the framework of the *Local Multi-Platform Linear Algebra Toolbox* (lmpLAtoolbox) that forms the basis of the parallel and open-source HiFlow^3 FEM software package [1] developed at EMCL. The main goal of the lmpLAtoolbox is to provide a complete, generic and robust set of routines across a multitude of parallel platforms. It provides maximal flexibility to the developer and refrains users from in-depth hardware knowledge while delivering best performance and optimal usage of computing resources. The modular setup of our approach offers a high-level of abstraction by providing unified interfaces for basic matrix and vector routines across diverse platforms. Only a single code base is required for a portable implementation of highly efficient numerical schemes. The same code can be run on several hardware platforms where the final choice of platform can be taken at run time. Platform-specific implementations are transparent to the user. Therefore, linear and non-linear solvers as well as preconditioners can be implemented easily and generically without any information on the underlying hardware platform while keeping platform-adapted and tuned code.

HiFlow^3 is a multi-purpose finite element package for solving a wide range of problems modeled by PDEs. The software stack is based on modular techniques which provide a generic and flexible approach. It is based on object-orientation in C++. 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 data among the nodes and performs cross-node computations. The local level (local multi-platform LAtoolbox) takes care of the on-node routines offering a unified interface independent of the specific platform. Figure 2 presents a diagram of the structure of the LAtoolbox in HiFlow^3 detailing the hierarchy of the lmpLA-toolbox.

The lmpLAtoolbox contains two basic C++-classes for vectors and matrices. The vector object contains the vector data and provides all vector routines (e.g. vector update, scaling, rotation, scalar product and others). Similar, the matrix object provides all matrix routines (matrix-vector multiplication, matrix-matrix multiplication, scaling functions and others). After the discretization of the PDE by means of finite element or related methods a sparse linear system is obtained. Therefore, the matrix data in the object is stored in a sparse structure. Currently, the module supports compressed sparse row (CSR) format [5], but the library can be easily extended by other formats.

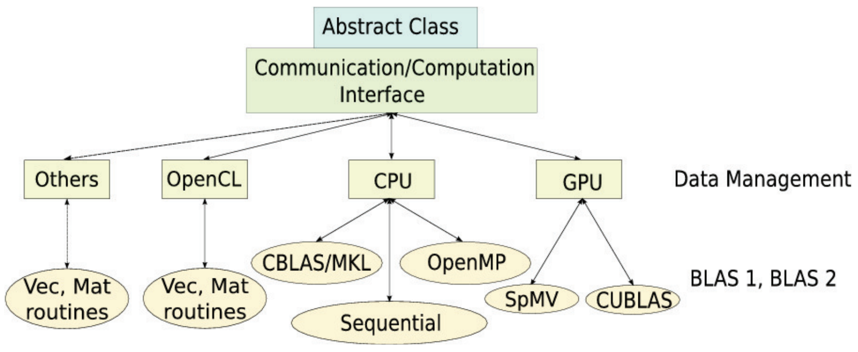

Fig. 2. Structure of the *Local Multi-Platform Linear Algebra Toolbox* (lmpLAtoolbox)

Each matrix/vector class has its base class and is inherited by a platform-management class which takes care of the memory allocation, data placement, and data access on all platforms. Each particular and platform-specific implementation is inherited from the data management class. The base class for each vector and matrix object provides a complete interface for all routines and operations. A typical usage of the library relies on using only pointers to the base classes. A code example is presented in Algorithm 7. No specific platform has to be specified in the code. The final decision on the actual platform and implementation can be taken at run time by means of user input or any library-guided choice.

Algorithm 7 Example code for the lmpLAtoolbox

```
// Declare a CPU matrix object
CPU_lMatrix<double> mat_cpu;
// Declare matrix and vector pointers to the base class
lMatrix<double> *mat;
lVector<double> *x, *y ;
int colors, *c_sizes, *perm;

// Read a cpu matrix from a mtx file
mat_cpu.ReadFile('matrix.mtx');

// Multi-coloring permutation of the matrix
mat_cpu.Multicoloring(colors, &c_sizes, &perm);
mat_cpu.Reorder(perm);

// initalize empty matrix on a specific platform
// (nnz,nrow,ncol,name,platform,implementation,format)
mat=init_matrix<double>(0, 0, 0, "A", platform, impl, CSR);

// Copy the sparse structure
mat->CopyStructureFrom(mat_cpu);

// Copy only the values of the matrix
mat->CopyFrom(mat_cpu);

delete mat_cpu;

// initialize vector x for a specific platform and implementation
x=init_vector<double>(size, "vec x", platform, impl);
// create vector y as x (clone it)
y=x->CloneWithoutContent();

// init some values in the vector x
x->SetBlockValues(0, size, values);

// Usage of BLAS 1 and BLAS 2 routines
y->CopyFrom(*x); // y = x
y->Axpy(*x, 4.3 ); // y = y + 4.3*x
x->Scale(1.2); // x = 1.2 * x
mat->VectorMult(*y, x); // x = mat*y

std::cout << y->dot(*x); // scalar product
delete x, y, mat;
```

Each matrix and vector object can perform the following operations via its base class interface:

– Perform vector/matrix routines with itself or objects on the same platform (not necessarily with the same implementation)
– Clone itself, i.e. create an object on the same platform with the same implementation
– Copy its data (matrix/vector) to another object on the same or different platform and/or implementation
– Advanced data manipulation techniques like vector splitting, concatenating, data extraction based on irregular patterns and others
– Classes for CPUs contain routines for pre-processing like I/O to files, incomplete LU factorization, or graph analysis (e.g. multi-coloring, level-scheduling, (reverse) Cuthill-McKee ordering)

With these interfaces inherited from the base class iterative solvers and preconditioners can be easily built without hard-coding a specific platform or implementation. Using this kind of abstraction the same source code can be used on all platforms. Like that we provide a generic, robust, flexible, and extendable approach for building applications. In this way, the developer has no direct access to the raw data of matrices or vectors, but by declaring a platform-specific object the user can obtain direct pointers to the data of its object. These techniques can be used for advanced platform-specific algorithms, like special nested loop iterations or irregular data access.

Parallelization of the routines is based on fine-grained data-parallel techniques. For implementations on multi-core x86 CPUs our library provides a purely sequential version and two parallel versions: CBLAS/MKL routines and OpenMP parallel routines. For our GPU backends we support NVIDIA GPUs where the vector routines are based on CUBLAS, and for the matrix-vector operations we have implemented our own kernels. We use scalar versions of the data-parallelization (i.e. a single thread for processing a matrix row) due to the fact that the considered FEM matrices usually have a very low number of non-zero elements with short inner loops. Details about the parallelization techniques for NVIDIA GPUs can be found in [7, 6]. Currently, we are working on an OpenCL [25] implementation for different backends like multi-core CPUs, NVIDIA and ATI GPUs and the STI Cell BE.

Currently, we provide Krylov subspace solvers, namely CG and GMRES. Newton-like methods are used for the non-linear solvers. The preconditioners based on specific matrix partitionings and re-orderings (e.g. multi-coloring and level-scheduling) are also built via our unified interfaces. An example for parallel LU substitutions for the symmetric Gauß-Seidel preconditioning can be found in Algorithm 8. Due to the fact that the block-diagonal sub-matrices D_i contain only diagonal elements they are stored as vectors and element-wise vector-vector routines are applied. All of the routines in the preconditioner building step are performed on the CPU – like graph analysis, re-ordering and incomplete factorizations. After that all of the necessary matrices and vectors for solving the preconditioned equation are allocated on the selected platform and via pointers they are upcasted to their base classes. Doing so, we can apply the preconditioning step by simply performing the matrix/vector and vector/vector routines by calling the corresponding interface of the base class – see for example the symmetric Gauss-Seidel solver described in Algorithm 8.

The usage of the lmpLAtoolbox within an MPI environment on distributed memory architectures can be combined with all current platforms and implementations. An advanced data blocking technique is used in the GPU class for minimizing the effect of the communication bottleneck over the PCIe bus [23]. Due to the lack of an appropriate preconditioning algorithm for distributed memory architectures, our preconditioners are currently restricted to node-level execu-

Algorithm 8 LU-sweeps for solving $Mz = r$ for symmetric Gauss-Seidel with preconditioning matrix $M := (D + L)D^{-1}(D + R)$

```
Split and copy vector $r$ into $z_i$ for $i = 1, \dots, B$
Forward step $z_i := D_i^{-1}(z_i - \sum_{j=1}^{i-1} L_{i,j} z_j)$ for $i = 1, \dots, B$
Diagonal step $z_i := D_i z_i$ for $i = 1, \dots, B$
Backward step $z_i := D_i^{-1}(z_i - \sum_{j=1}^{B-i} R_{i,j} z_{i+j})$ for $i = 1, \dots, B$
Concatenate vector $z_i$ into $z$
```

tion, i.e. to multi-socket multi-core processors on a shared memory node or to single GPUs. Extensions, also for multi-GPU and heterogeneous configurations, are in the test phase.

5 Effects of Re-ordering on Matrix Sparsity Patterns

In this section we consider the impact of matrix re-ordering techniques on the structure and sparsity pattern of the system matrix. On modern multi-core and manycore architectures like cache-based multi-core CPUs and processors with user-managed local memory (like shared memory on GPUs) data locality is a performance-critical issue. The matrix decomposition into blocks with a lower bound on the block size is a necessary building block for fine-grained parallel methods like our preconditioners. 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.

By means of small test matrices we investigate how multi-coloring, level scheduling and the power(q)-pattern enhanced multi-colored ILU(p,q) method is influencing the number of blocks for parallel execution (i.e. the number of colors) and the sparsity pattern of the system matrix. In case of the ILU decomposition also the factorized matrices are 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 chosen for a proper visualization of the matrix patterns in spy plots. In Section 6 performance analysis is conducted for large matrices. Smaller matrices – as considered in this section – are not an appropriate input for fine-grained parallel preconditioners since lower bounds for block sizes in matrix decompositions are mostly missed.

The symmetric and positive definite `nos5` matrix of size 468-by-468 and 2820 non-zero elements describes a finite element approximation of beams by a biharmonic operator with one end free and one end fixed [27]. The symmetric and positive definite `gr3030` matrix is derived from a finite difference discretization of a Laplace problem. It has dimension 900-by-900 with 4322 non-zero entries [28]. The original sparsity patterns are shown in Figure 3 with `nos5` on the left and `gr3030` on the right.

In this section we consider the conjugate gradient method (CG) [31] as an iterative Krylov subspace-type solver. We use right hand side set to one and initial guess zero. For the `nos5` matrix it takes more than 400 iterations to achieve a relative residual smaller than 10^{-6} as shown in Figure 4 (left). With the multi-colored symmetric block Gauß-Seidel (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 by a factor 71. The ILU(p) efficiency with respect to iteration count is increasing with p. A similar observation is made for

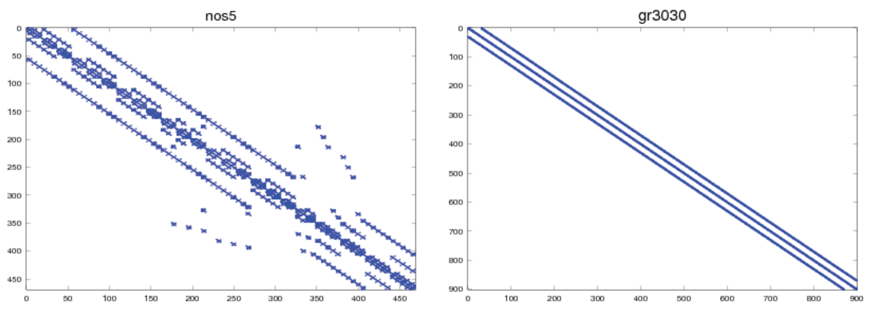

Fig. 3. Sparsity patterns of the `nos5` matrix of size 468-by-468 with 2820 non-zero elements (left) and the `gr3030` matrix of size 900-by-900 with 4322 non-zero elements (right).

the `gr3030` matrix in Figure 4 (right). The preconditioners decrease the number of iterations, where ILU(3) has an acceleration factor of 4.7, i.e. 4.7 times less iterations are necessary with the ILU(3) preconditioner

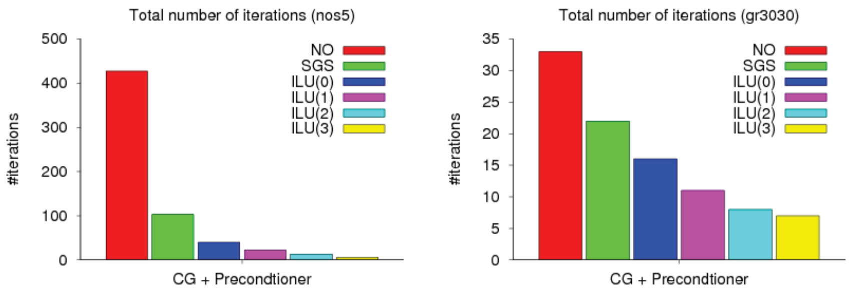

Fig. 4. Level scheduling: Number of CG iterations without preconditioner and with level scheduling for symmetric Gauß-Seidel (SGS) and ILU(p) preconditioners for $p = 0, 1, 2, 3$ for the `nos5` matrix (left) and the `gr3030` matrix (right).

In Figure 5 the improvements with respect to CG iteration count are presented for the power(q)-pattern enhanced multi-colored ILU(p,q) preconditioners with and without drop-off. 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 with increasing p. For the results with drop-off, i.e. $q < p+1$, efficiency is slightly worse.

When level-scheduling is applied to the factorized matrices L_p and U_p of the ILU(p) decomposition of the `nos5` matrix, i.e. $A = L_pU_p + R_p$ with p-fill-ins, we obtain 39 levels for $p = 0$, 136 levels for $p = 1$, 312 levels for $p = 2$, and 403 levels for $p = 3$. Accordingly, each block only contains a very low number of elements (only one element in many cases). The necessary degree of parallelism is not given in this scenario. The matrix patterns obtained by applying level scheduling renumbering 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 6 for the `nos5` matrix. 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 7 are observed. After renumbering, this structure of A is used within

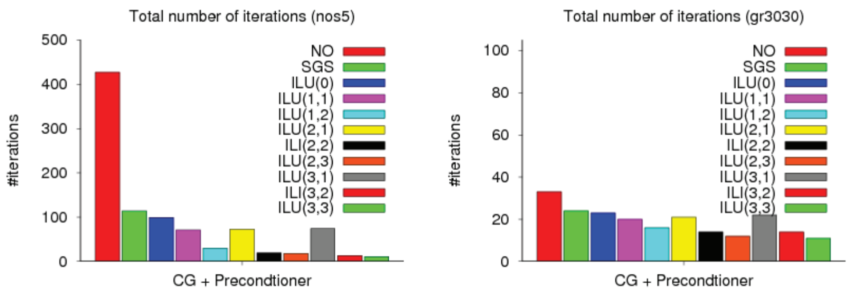

Fig. 5. Power(q)-pattern enhanced multi-colored ILU(p,q) preconditioners: Number of CG iterations without preconditioner and with multi-colored symmetric Gauß-Seidel (SGS) and power(q)-pattern enhanced multi-colored ILU(p,q) preconditioners for $p = 0, 1, 2, 3$ with and without drop-off for the `nos5` matrix (left) and the `gr3030` matrix (right).

parallel matrix-vector operations in the parallel CG solver. These figures show how locality is affected by the re-ordering of nodes.

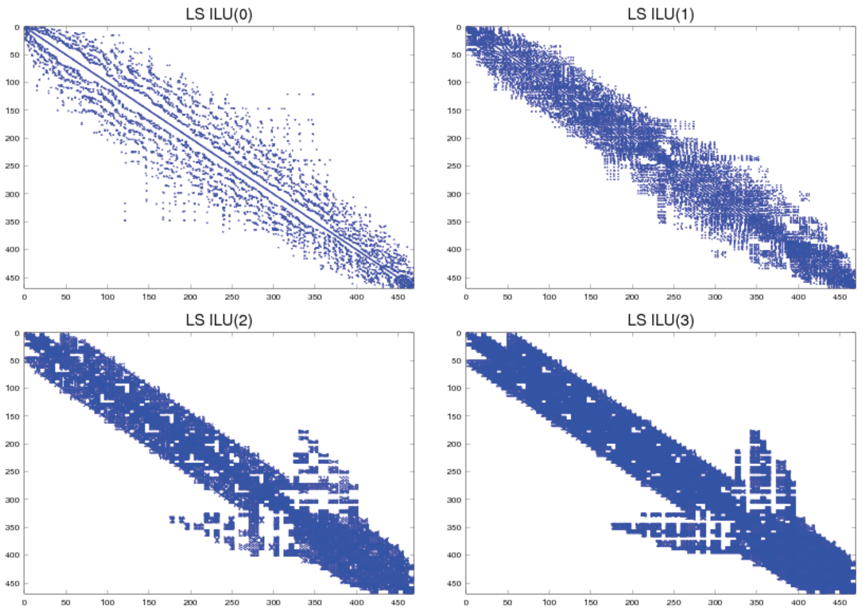

Fig. 6. `nos5`: Level scheduling re-ordering π_p applied to the factorized matrices L_p and U_p (combined in a single matrix structure) given by the ILU(p) decomposition $A = L_pU_p + R_p$ with level-p fill-ins.

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 level scheduling re-ordering π_p applied is shown in Figure 8. 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 9 are observed.

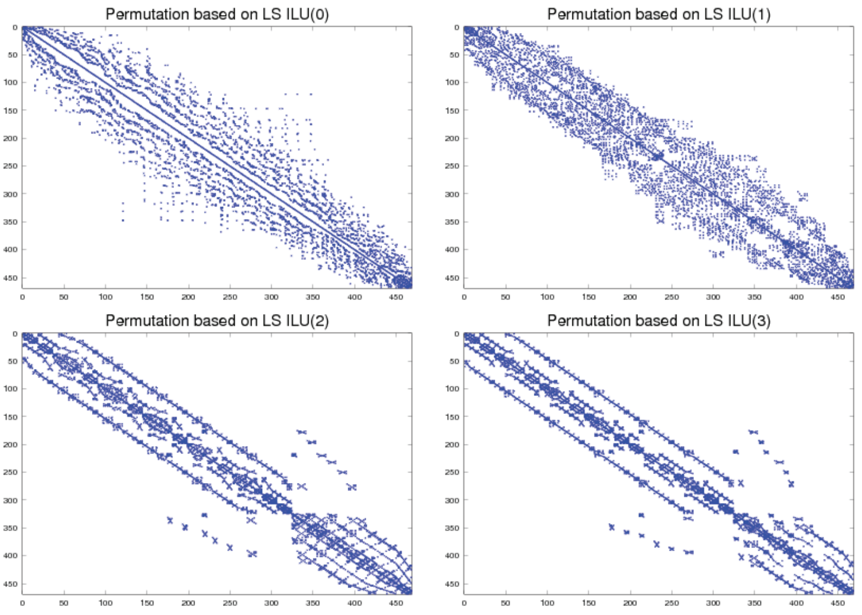

Fig. 7. `nos5`: Level scheduling re-ordering π_p corresponding to ILU(p) applied to the original matrix A.

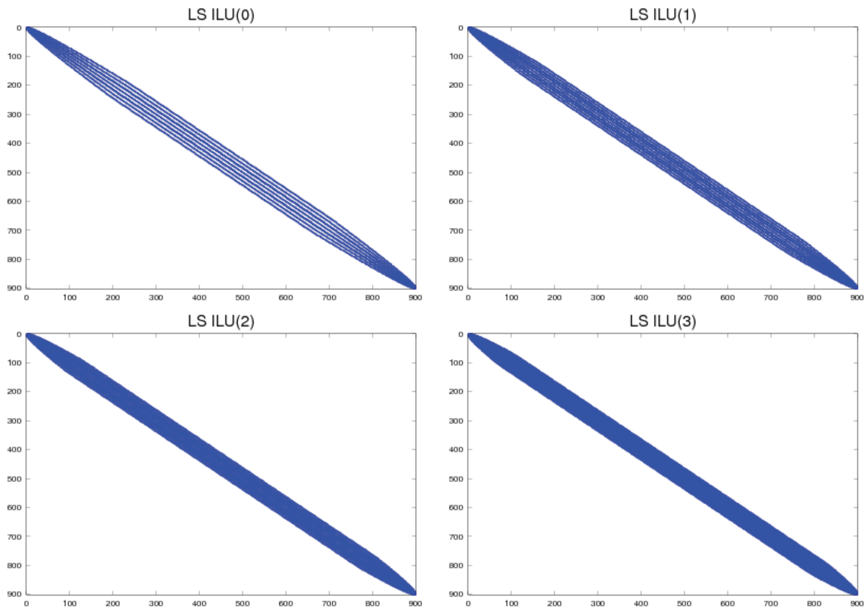

Fig. 8. `gr3030`: Level scheduling re-ordering π_p applied to the factorized matrices L_p and U_p (combined in a single matrix structure) given by the ILU(p) decomposition $A = L_pU_p + R_p$ with level-p fill-ins.

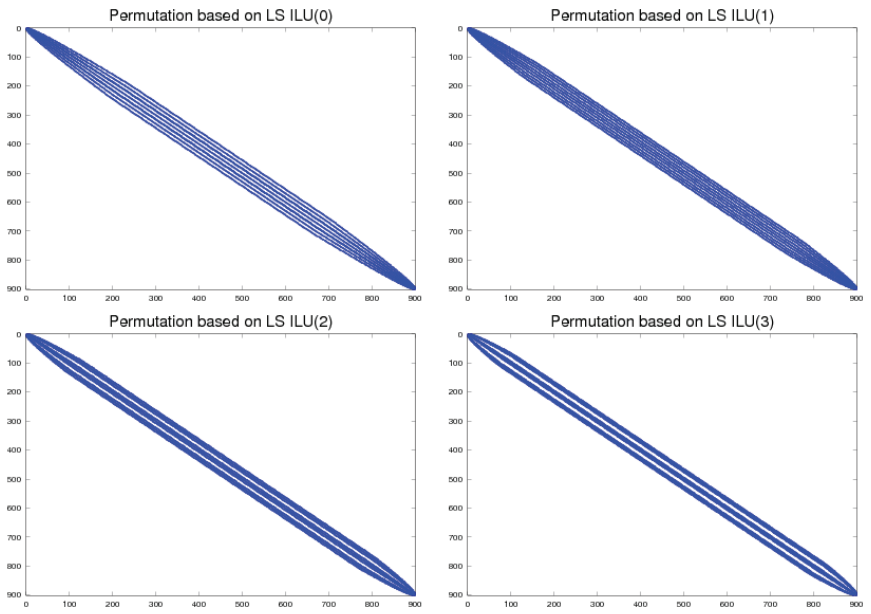

Fig. 9. `gr3030`: Level scheduling re-ordering π_p corresponding to ILU(p) applied to the original matrix A.

A higher degree of parallelism can be obtained by applying multi-coloring techniques to the powers $|A|^q$ of the modulus of the system matrix. For the `nos5` matrix and $q = 1, 2, 3, 4$ we observe 9, 33, 84 and 157 colors. The corresponding multi-color 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 10. These matrices are the starting point for the incomplete factorizations. In this setting $\pi_1 A \pi_1^{-1}$ (left upper figure) is the superset for the ILU(0,1) decomposition (see upper left figure in Figure 11).

For the power(q)-pattern enhanced multi-colored ILU(p,q) preconditioner, Figure 11 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 upper sub-figures show the factorized multi-colored ILU(p) decomposition for $p = 0$ and $p = 1$ with permutation based only on the original matrix, i. e. $|A|$. The resulting matrix in the left figure is equal to the original ILU(0) decomposition. For the matrix shown in the right figure the described drop-off technique is applied. The first figure of the second row presents ILU(1,2), the factorization of first order without drop-offs. The second figure in this row shows a drop-off strategy for $p = 2$ and $q = 1$ using only 9 colors. Results for ILU(2,q) are shown in the last row of Figure 11 for $q = 2$ and $q = 3$ resulting in 84 and 157 colors, where the latter case is without drop-off. Decomposition patterns for ILU(3,q) for the `nos5` matrix are depicted in Figure 12 for $q = 1, 2, 3, 4$. The drop-off strategy is applied for $q = 1, 2, 3$ for reducing the number of colors which is 9, 33, 84 and 157.

The 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-color permuted system matrix, i.e. $\pi_q A \pi_q^{-1}$, for the `gr3030` matrix is depicted in Figure 13. Here again, π^q is obtained from multi-color analysis of $|A|^q$. We find 4, 9, 16 and 25 colors for $q = 1, 2, 3, 4$.

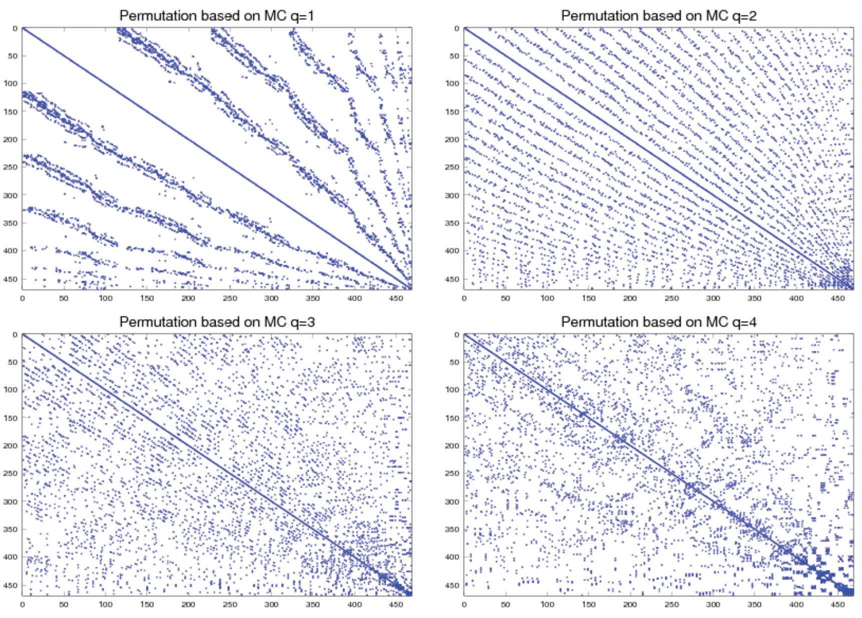

Fig. 10. `nos5`: Sparsity patterns of the permuted linear system $\pi_q A \pi_q^{-1}$ with multi-coloring permutation π_q obtained from analysis of $|A|^q$ for $q = 1, 2, 3, 4$ with 9, 33, 84, and 157 colors.

For the ILU(p,q) method, Figure 14 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 `gr3030` matrix. The upper sub-figures show the multi-colored factorization decomposition for $p = 0$ and $p = 1$ with $q = 1$ with 4 colors. The left figure of the second row shows the ILU(1,2) permutation, the right figure of 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 elements 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 15. The drop-off strategy is applied for $q = 1, 2, 3$ for reducing the number of colors which is 4, 9, 16 and 25.

6 Performance Analysis of Parallel Preconditioners

In this section we investigate and present efficiency and scalability of our described preconditioners in terms of reduced number of iterations, in terms of solver times, in terms of parallel speedup, and in terms of 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 for p and q, where $q = p + 1$ represents the natural scenario and $q < p + 1$ represents the drop-off technique with a reduced number of colors where non-diagonal elements in the diagonal blocks are deleted.

Our test suite 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 to model 2D animal/gene movement flow [34]. The `s3dkq4m2` matrix is obtained from a finite element analysis of cylindrical shells on a uniform quadrilateral mesh [29]. The `g3_circuit` matrix results from a circuit simulation problem [35]. In Table 1 basic data and properties of the test matrices are listed. It shows the numbers

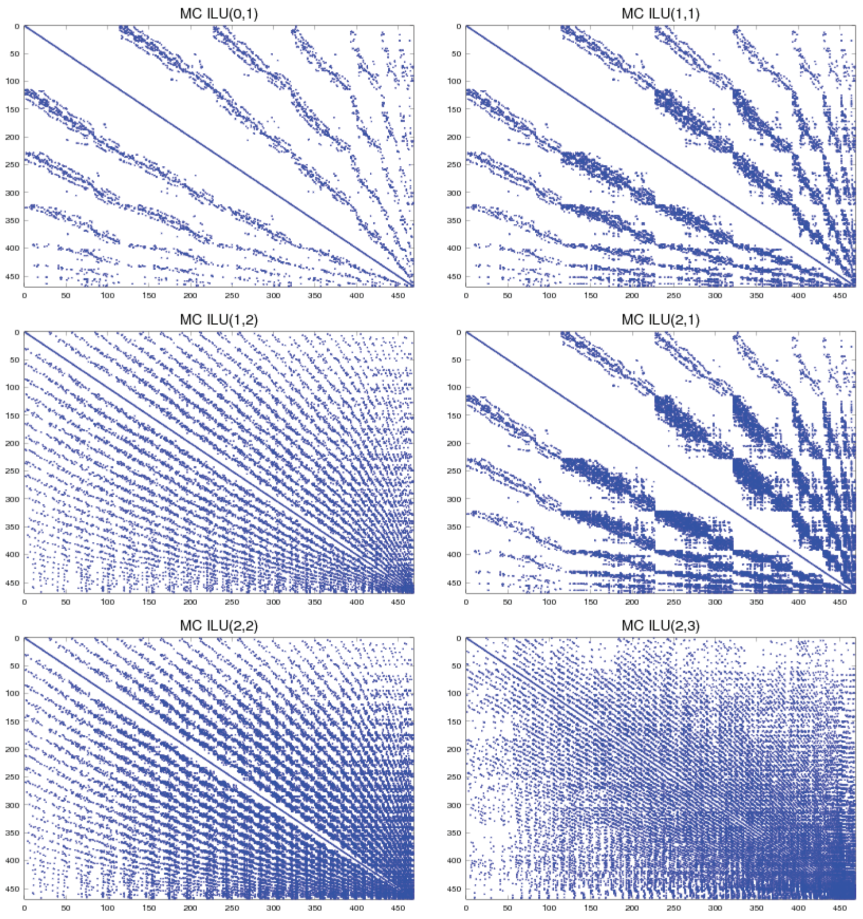

Fig. 11. `nos5`: Sparsity patterns for power(q)-pattern enhanced multi-colored ILU(p,q) decomposition with and without drop-off; upper row: ILU(0,1) (no drop-off) and ILU(1,1) (drop-off), second row: ILU(1,2) (no drop-off) and ILU(2,1) (drop-off); last row: ILU(2,2) (drop-off) and ILU(2,3) (no drop-off).

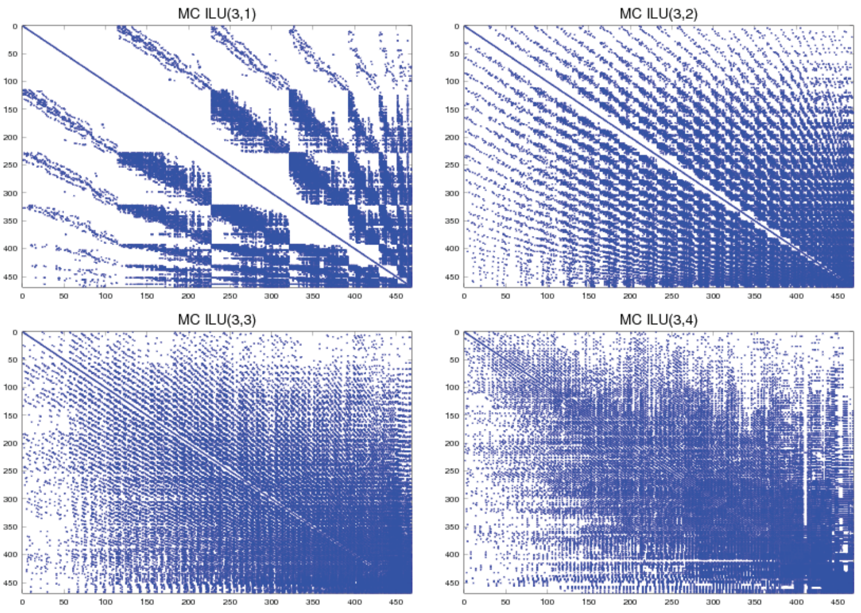

Fig. 12. `nos5`: Sparsity patterns for the power(q)-pattern enhanced multi-colored ILU(3,q) decomposition with drop-off ($q = 1, 2, 3$) and without drop-off ($q = 4$); upper row ILU(3,1) and ILU(3,2); lower row: ILU(3,3) and ILU(3,4).

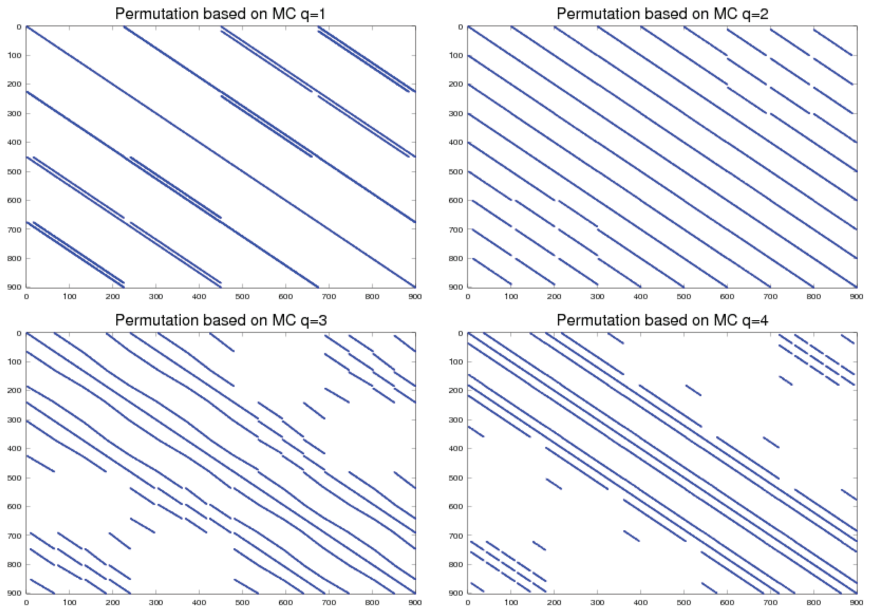

Fig. 13. `gr3030`: Sparsity patterns of the permuted linear system $\pi_q A \pi_q^{-1}$ with multi-coloring permutation π_q obtained from analysis of $|A|^q$ for $q = 1, 2, 3, 4$ with 4, 9, 16, and 25 colors.

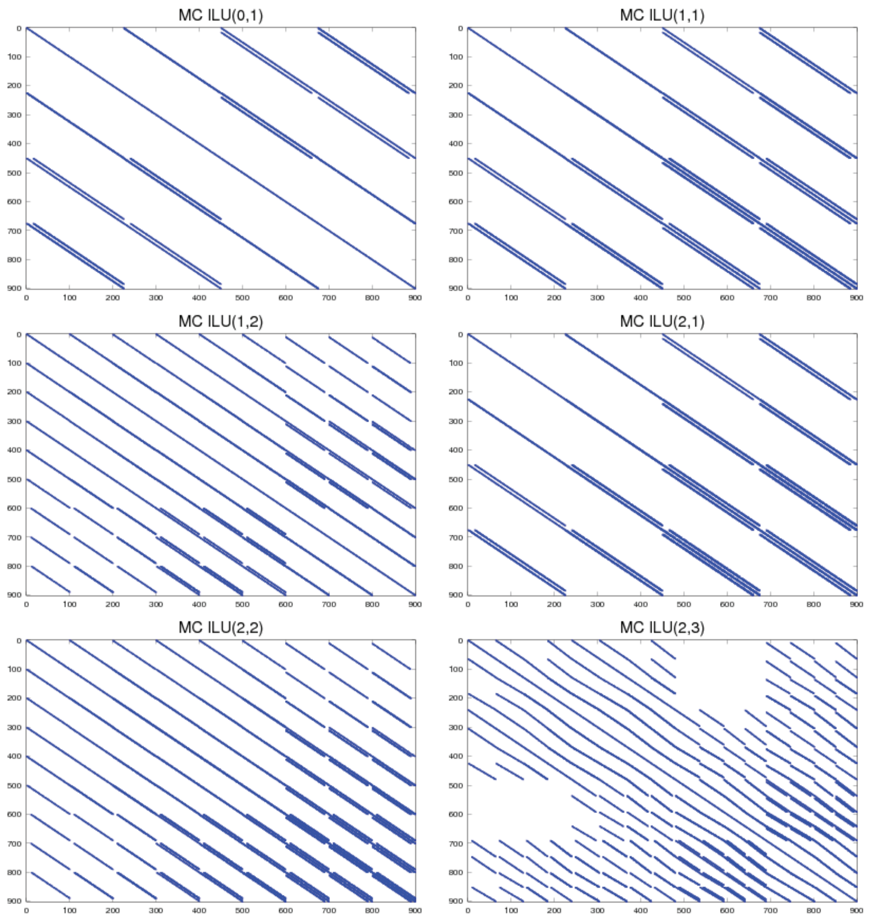

Fig. 14. `gr3030`: Sparsity patterns for power(q)-pattern enhanced multi-colored ILU(p,q) decomposition with and without drop-off; upper row: ILU(0,1) and ILU(1,1); second row: ILU(1,2) and ILU(2,1); last row: ILU(2,2) and ILU(2,3).

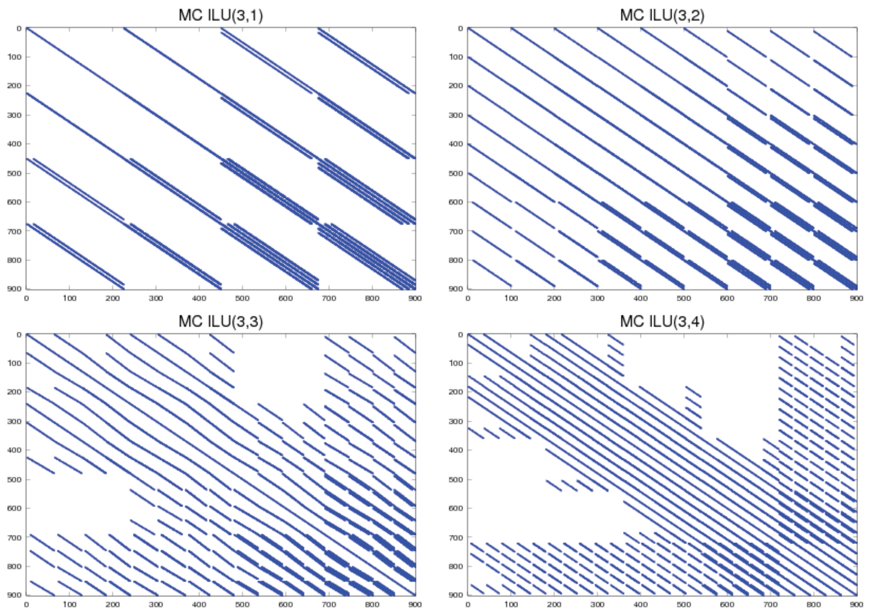

Fig. 15. gr3030: Sparsity patterns for power(q)-pattern enhanced ILU(3, q) decomposition with and without drop-off; upper row ILU(3,1) and ILU(3,2), lower row ILU(3,3) and ILU(3,4).

of rows, the number of columns and the number of colors when multi-coloring is applied to the original matrix. In the last column, the number of sparse matrix-vector operations with respect to the block decomposition is given for the multi-colored ILU(0) preconditioner. For the `g3_circuit` matrix the decomposition into colors (by applying multi-coloring to the original matrix) is imbalanced with 689390, 789436, 106502 and 150 entries per color. For the `s3dkq4m2` and `ecology2` matrix the block distributions have balanced sizes. In general, smaller matrices (like `3dkq4m2`) are better suited for the cache-oriented CPUs since 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.

Name	Description of the problem	#rows	#non-zeros	#colors	#block-SpMV in MC-ILU(0)
`ecology2`	Animal/gene movement	999999	4995991	2	2
`s3dkq4m2`	Cylindrical shells	90449	4820891	24	552
`g3_circuit`	Circuit simulation	1585478	7660826	4	12

Table 1. Description and properties of considered test matrices.

Our experiments are performed on a hybrid test platform, a dual-socket Intel Xeon (E5450) quad-core system (eight cores in total) that is accelerated by an NVIDIA Tesla S1070 GPU system with four GPUs attached pairwise by PCIe to one socket each. The memory capacity of a single CPU and GPU device is 16GB and 4GB respectively.

For our performance analysis we are using the CG solver as an iterative Krylov subspace solver for symmetric positive definite matrices. As stopping criterion we choose 10^{-6} for the relative residual. The initial guess for the iterative solver is taken to be zero and the right hand side is initialized with constant value one.

For some GPU kernels performance can be improved by using texture caching, i.e. replacing loads from device memory by texture fetches which is a viable alternative to explicit pre-caching with shared memory. For some matrices, we see improvements, for others we find severe drawbacks from texture caching due to call overheads. For running the CG method without the preconditioning step we use texture caching for the matrix-vector multiplications but for the preconditioned CG solver we disable texture caching. Details on the impact of texture caching on multi-coloring sweeps is presented in [22].

Table 2 summarizes the iteration counts for the CG solver for the three test matrices for achieving the prescribed error tolerance. 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 amount consumed by the preconditioning step. i.e. the solution of the block-triangular systems. Moreover, Table 2 details the number of colors in the multi-colored SGS and ILU(0) scheme, in the power(q)-pattern enhanced multi-colored ILU(p,p+1) scheme, and in the drop-off version ILU(p,q) for $p = q = 3$. In comparison to these numbers, the number of levels in the level scheduling algorithm applied to the original matrix, i.e. the number of blocks, is 2593 for `ecology2`, 2388 for `s3dkq4m2`, and 1998 for `g3_circuit`.

		No precond	SGS	ILU(0)	ILU(1,2)	ILU(2,3)	ILU(3,4)	ILU(3,3)
`ecology2`	# its	5391	2783	2855	1815	1308	997	1277
	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	12760	1328	1242	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	535056	12728	3918	2203	1600	965	6086
	acc. fact.	1.0	42.0	136.5	242.8	334.4	554.4	87.9
	# colors		24	24	56	96	150	96

Table 2. 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 16 details the performance of the preconditioner for the `ecology2` problem. The left figure shows the number of iterations with and without preconditioner for reaching the prescribed error tolerance. We consider the multi-colored SGS preconditioner with two colors, the multi-colored ILU(0) decomposition with two colors, and the power(q)-pattern method enhanced multi-colored ILU(p,q) method with level-p fill-ins for $p = 1, 2, 3$. 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 right 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 determination). It depicts solver times for the sequential solution running on a single core, the OpenMP parallel solution running on eight cores, and the single GPU version with texture caching for the non-preconditioned solver and without texture caching for the preconditioned one.

In Figure 17 the associated speedups for the `ecology2` matrix are presented. The left figure shows the parallel speedup 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 eight CPU cores. The right figure shows the overall acceleration of the preconditioned solver over the non-preconditioned version with fixed hardware and implementation. For this test problem, the preconditioner on the GPU accelerates the solver by a factor of up to 1.7. In some cases, there is 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.

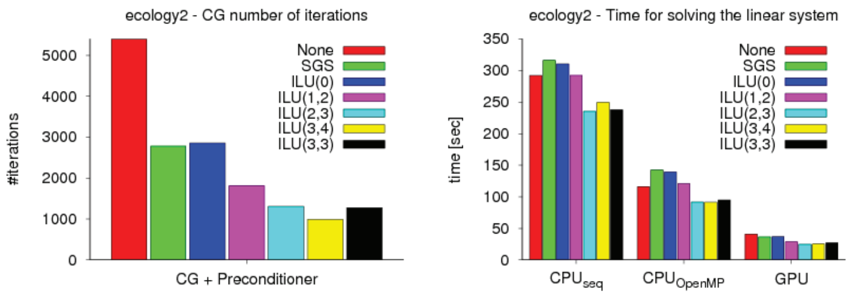

Fig. 16. `ecology2 matrix`: Iteration numbers (left) and solver time (right) for multi-colored symmetric Gauß-Seidel (SGS), multi-colored ILU(0), and power(q)-pattern enhanced ILU(p,q) with and without drop-off on single/eight core multi-core CPU(s) and a single GPU.

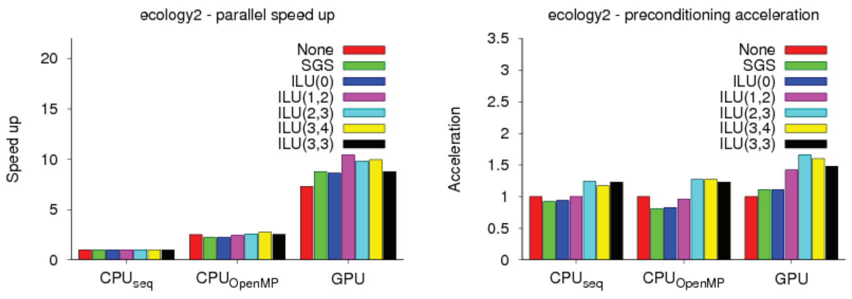

Fig. 17. `ecology2` matrix: Parallel speedup (left) and acceleration factors by preconditioning step (right) for multi-colored symmetric Gauß-Seidel (SGS), multi-colored ILU(0), and power(q)-pattern enhanced ILU(p,q) with and without drop-off on single/eight core multi-core CPU(s) and a single GPU.

In Figure 18 the performance of the preconditioner for the `g3_circuit` matrix is shown. The left figure shows the number of iterations with and without preconditioner. We find 4 up to 35 colors for this problem. The multi-colored SGS and the multi-colored ILU(0) decomposition have 4 colors whereas the power(q)-pattern method enhanced multi-colored ILU(3,4) has 35 colors. All preconditioners show a significant decrease of the iteration count (left) and solver time (right). In Figure 19 (left) we find parallel speedups between 2 to 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. The right figure shows the acceleration factor of the preconditioner with respect to time for fixed hardware and implementation. The total acceleration factor is between 4 and 9 for this test problem. Acceleration and parallel speedup is observed for all presented preconditioner configurations. The drop-off technique for ILU(3,3) reduces the number of colors from 35 to 17. It gives better results with respect to the solver time and better acceleration factors.

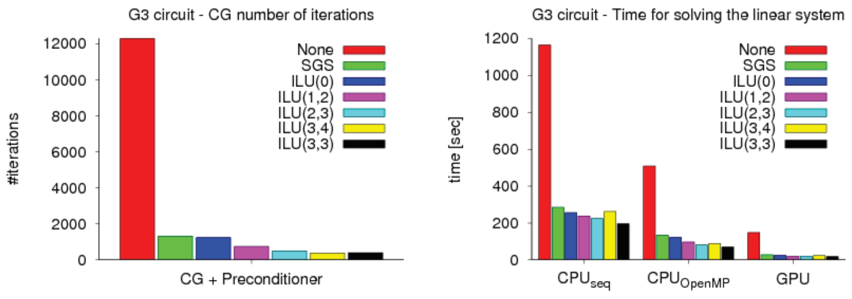

Fig. 18. `g3_circuit` matrix: Iteration numbers (left) and solver time (right) for multi-colored symmetric Gauß-Seidel (SGS), multi-colored ILU(0), and power(q)-pattern enhanced ILU(p,q) with and without drop-off on single/eight core multicore CPU(s) and a single GPU.

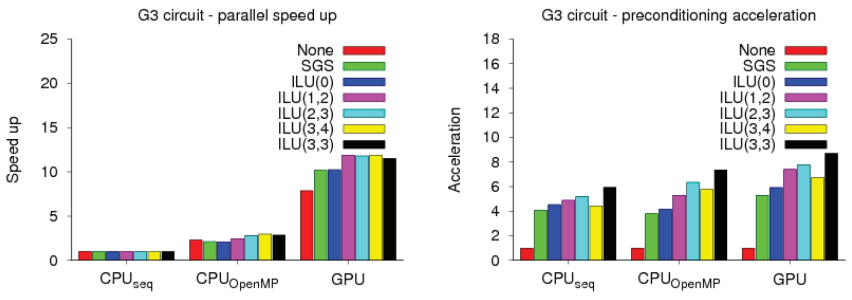

Fig. 19. `g3_circuit` matrix: Parallel speedup (left) and acceleration factors by preconditioning step (right) for multi-colored symmetric Gauß-Seidel (SGS), multi-colored ILU(0), and power(q)-pattern enhanced ILU(p,q) with and without drop-off on single/eight core multi-core CPU(s) and a single GPU.

Figure 20 depicts performance data for the `s3dkq4m2` matrix. The left figure shows a zoom-in plot for the iteration count where the left-most bar for the non-preconditioned case is cut off in both figures. The number of iterations for the non-preconditioned CG is 535056 in this example. It shows that the number of iterations without preconditioner is massive but can be decreased considerably by preconditioning. Figure 20 (left) shows 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 multi-colored

ILU(0) preconditioner, see Figure 20 (right). The drop-off technique reduces the number of colors from 134 to 96 for $p = 3$ but has no positive effect with respect to efficiency. Solver times are improved by the parallel OpenMP and GPU versions only for $p = 0, 1, 2$, with best results for $p = 0$. As shown in Figure 21 (left) the OpenMP parallel speedup is not much more than two and gets worse when increasing p. For $p = 2$ or larger, results on the GPU break down when texture caching is enabled (not shown). Due to the large number of colors and the small matrix size the blocks per color become too small and the overhead from texture caching is dominating the solver time. The maximal speedup on the GPU is around three and about 50% higher than the eight core OpenMP parallel speedup. In the parallel case, the acceleration factor from preconditioning is still immense; see Figure 21 (right). Compared to the sequential case, the preconditioner does not lose much of its efficiency. For ILU(0) the acceleration factor is more than 60 on the GPU without texture caching. 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 reduced number of colors in ILU(3,3) gives bad results.

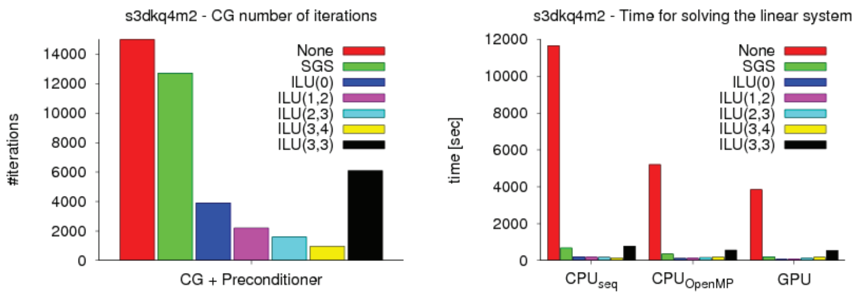

Fig. 20. `s3dkq4m2` matrix: Iteration numbers (left, zoom-in with bar for non-preconditioned case cut off, #its(no precond)=535056) and solver time (right) for multi-colored symmetric Gauß-Seidel (SGS), multi-colored ILU(0), and power(q)-pattern enhanced ILU(p,q) with and without drop-off on single/eight core multi-core CPU(s) and a single GPU.

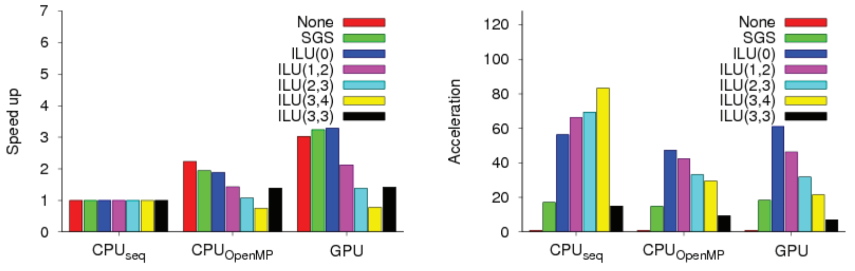

Fig. 21. `s3dkq4m2` matrix: Parallel speedup (left) and acceleration factors by preconditioning step (right) for multi-colored symmetric Gauß-Seidel (SGS), multi-colored ILU(0), and power(q)-pattern enhanced ILU(p,q) with and without drop-off on single/eight core multi-core CPU(s) and a single GPU.

7 Future Work

The status of our current work still gives room for further improvements and investigations. In our current experiments we have treated the setup of the preconditioner as a preprocessing step. In future work we will present optimization approaches and run time considerations for this setup step. A model for the overall preconditioning costs, including building of the sparsity pattern of $|A|^{p+1}$, will be included. Additional optimization potential has been identified for matrix-vector multiplications in case of small matrices and for the forward and backward substitutions. Pivoting strategies on the level of color classes are an inevitable means to improve stability of our preconditioners. Moreover, we are identifying the class of matrices for which the original ILU(p) scheme in Algorithm 3 gives the same factorization as our modified ILU(p,q) scheme in Algorithm 5.

As the lmpLAtoolbox supports code execution on multi-GPU systems and for heterogeneous configurations with different types of processors contributing to the parallel solution, our preconditioners will be tested for these scenarios. With respect to further scalability we are working on concepts for preconditioners on distributed memory architectures where the global MPI layer in our LAtoolbox will be used.

In further research we are investigating non-symmetric systems with preconditioned GMRES solvers [31]. Furthermore, additional preconditioners will be included into the lmpLAtoolbox and the HiFlow^3 package. Besides the preconditioners based on LU-sweeps, we are currently working on the approximate inverse preconditioners [31]. Moreover, we are investigating properties of the FSAI algorithm [26] and try how to determine proper sparsity patterns. In addition, we are trying to deploy the preconditioning techniques in the context of parallel smoothers for multi-grid methods with promising results obtained so far.

8 Conclusion and Outlook

In this work we have considered node-level parallel preconditioners for shared memory based multi-core systems with several sockets and for GPU-enhanced systems. We have investigated matrix re-ordering schemes for introducing scalable parallelism for block-decomposed preconditioners based on additive or multiplicative matrix splittings. Our focus was on parallel solution of resulting triangular systems originating from Gauß-Seidel-type methods and incomplete LU decompositions. Level scheduling and multi-coloring methods have been used to generate block decompositions where the diagonal blocks are diagonal itself for easy inversion.

We have presented a new parallel algorithm for performing ILU(p) with level p fill-ins by means of the *power(q)-pattern enhanced multi-colored ILU(p,q) method*. The major advantage is that multi-coloring can be applied to this new structure before performing the ILU(p) decomposition with additional fill-ins. The diagonal structure of the block diagonals is then preserved and fill-ins only occur outside of the blocks on the diagonal. The fill-in positions are known in advance by precomputing a superset of the data distribution pattern. Thereby, we eliminate costly insertion of new matrix elements into dynamic data structures. We have reported speedups in parallel environments as well as convincing acceleration factors due to the parallel preconditioning techniques. In particular, efficiency of the preconditioner is maintained during parallel execution when compared to its sequential version.

Our approach provides a flexible, easy-to-handle and out-of-the-box preconditioner. We have outlined integration of the proposed preconditioners into the portable *lmpLAtoolbox* and the parallel finite element package HiFlow^3. In this context platform optimized solvers can be built by utilizing unified and generic interfaces. The choice of final hardware configuration can be taken at run time and there is no need for in-depth hardware knowledge for the user. Our presented results show considerable improvements. To the best of our knowledge this is the first time that a parallel version of ILU(p) preconditioners with fill-ins is successfully adapted and ported to GPU platforms that become increasingly important in the current and future computing landscape.

Acknowledgements

The Shared Research Group 16-1 received financial support by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and its collaboration partner Hewlett-Packard. We would like to express our gratitude to Felix Riehn and Niels Wegh who have implemented some parts of our software.

The Engineering Mathematics and Computing Lab (EMCL) at KIT has been awarded NVIDIA CUDA Center of Research for its research and efforts in GPU computing. The authors acknowledge the support by means of hardware donation from the company NVIDIA in that framework.

References

1. 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., Wilhelm, F.: HiFlow^3 – A Flexible and Hardware-Aware Parallel Finite Element Package (2010). URL http://www.emcl.kit.edu/preprints/emcl-preprint-2010-06.pdf
2. Asgasri, A., Tate, J.E.: Implementing the Chebyshev Polynomial Preconditioner for the Iterative Solution of Linear Systems on Massively Parallel Graphics Processors. In: CIGRE Canada Conference on Power Systems (2009)
3. Axelsson, O., Barker, V.A.: Finite element solution of boundary value problems : theory and computation. Computer science and applied mathematics
4. Balay, S., Brown, J., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page (2011). URL http://www.mcs.anl.gov/petsc
5. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., der Vorst, H.V.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. SIAM, Philadelphia, PA (1994)
6. Baskaran, M.M., Bordawekar, R.: Optimizing Sparse Matrix-Vector Multiplication on GPUs. Tech. rep., IBM (2009)
7. Bell, N., 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, pp. 1–11 (2009)
8. Bell, N., Garland, M.: CUSP: Generic parallel algorithms for sparse matrix and graph computations (2010). URL http://cusp-library.googlecode.com. Version 0.1.0
9. Bell, W.N., Olson, L.N., Schroder, J.B.: PyAMG: Algebraic multigrid solvers in Python v2.0 (2011). URL http://www.pyamg.org. Release 2.0
10. Braess, D.: Finite elements: theory, fast solvers, and applications in solid mechanics, 2. ed. edn. Cambridge Univ. Press, Cambridge (2001)

11. Brenner, S.C., Scott, L.R.: The mathematical theory of finite element methods, 2. ed. edn. Texts in applied mathematics. Springer, New York (2002)
12. Chen, D.: Analysis, implementation, and evaluation of vaidya's preconditioners (2001)
13. Chen, K.: Matrix preconditioning techniques and applications. Cambridge monographs on applied and computational mathematics ; 19
14. Ciarlet, P.G.: The finite element method for elliptic problems. Classics in applied mathematics; 40. Society for Industrial and Applied Mathematics, Philadelphia, PA (2002)
15. Cohen, J.: Presenatation slides (2011). URL http://www.ima.umn.edu/2010-2011/W1.10-14.11/activities/Cohen-Jonathan/Jonathan_Cohen_IMA.pdf
16. Demmel, J.W.: Applied numerical linear algebra. SIAM, Philadelphia, PA (1997)
17. Göddeke, D.: Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU cluster. Ph.D. thesis, Technische Universität Dortmund (2010)
18. Hackbusch, W.: Elliptic differential equations: theory and numerical treatment. Springer series in computational mathematics ; 18. Springer, Berlin (2003)
19. Henson, V.E., Yang, U.M.: BoomerAMG: a parallel algebraic multigrid solver and preconditioner. Appl. Numer. Math. **41**(1), 155–177 (2002)
20. Heroux, M., Bartlett, R., Hoekstra, V.H.R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., Williams, A.: An Overview of Trilinos. Tech. Rep. SAND2003-2927, Sandia National Laboratories (2003)
21. Heuveline, V., et al.: HiFlow^3 - Parallel Finite Element Software (2011). URL http://www.hiflow3.org/
22. Heuveline, V., Lukarski, D., Weiss, J.P.: Scalable multi-coloring preconditioning for multi-core CPUs and GPUs. In: UCHPC'10, Euro-Par 2010 Parallel Processing Workshops (2010)
23. Heuveline, V., Subramanian, C., Lukarski, D., Weiss, J.P.: A multi-platform linear algebra toolbox for finite element solvers on heterogeneous clusters. In: PPAAC'10, IEEE Cluster 2010 Workshops (2010)
24. Intel: Intel Math Kernel Library (MKL) (2011). URL http://software.intel.com/en-us/articles/intel-mkl/
25. Khronos Group: OpenCL (2011). URL http://www.khronos.org/opencl/
26. Kolotilina, L.Y., Yeremin, A.Y.: Factorized sparse approximate inverse preconditionings, I: theory. SIAM J. Matrix Anal. Appl. **14**, 45–58 (1993)
27. Matrix Market: gr_30_30 (2011). URL http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/laplace/gr_30_30.html
28. Matrix Market: nos5 (2011). URL http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos5.html
29. Matrix Market: s3dkq4m2 (2011). URL http://math.nist.gov/MatrixMarket/data/misc/cylshell/s3dkq4m2.html
30. Rudolf, F.: ViennaCL (2011). URL http://viennacl.sourceforge.net/. Version 1.1.1
31. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003)
32. Subramanian, C., Heuveline, V., Lukarski, D., Weiss, J.P.: Parallel preconditioning and modular finite element solvers on hybrid CPU-GPU systems. In: Proceedings of ParEng 2011 (2011)
33. Tuma, M., Benzi, M.: A comparative study of sparse approximate inverse preconditioners. Appl. Numer. Math **30**, 305–340 (1998)
34. University of Florida Sparse Matrix Collection: ecology2 (2011). URL http://www.cise.ufl.edu/research/sparse/matrices/McRae/ecology2.html
35. University of Florida Sparse Matrix Collection: G3_circuit (2011). URL http://www.cise.ufl.edu/research/sparse/matrices/AMD/G3_circuit.html