Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs

Vincent Heuveline

Dimitar Lukarski

Nico Trost

Jan-Philipp Weiss

No. 2011-09

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. 2011-09

Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs

Vincent Heuveline[1], Dimitar Lukarski[1,2*], Nico Trost[1] 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,
nico.trost@student.kit.edu

Abstract. Multigrid methods are efficient and fast solvers for problems typically modeled by partial differential equations of elliptic type. For problems with complex geometries and local singularities stencil-type discrete operators on equidistant Cartesian grids need to be replaced by more flexible concepts for unstructured meshes in order to properly resolve all problem-inherent specifics and for maintaining a moderate number of unknowns. However, flexibility in the meshes goes along with severe drawbacks with respect to parallel execution – especially with respect to the definition of adequate smoothers. This point becomes in particular pronounced in the framework of fine-grained parallelism on GPUs with hundreds of execution units. We use the approach of matrix-based multigrid that has high flexibility and adapts well to the exigences of modern computing platforms.

In this work we investigate multi-colored Gauß-Seidel type smoothers, the *power(q)-pattern enhanced multi-colored ILU(p)* smoothers with fill-ins, and *factorized sparse approximate inverse* (FSAI) smoothers. These approaches provide efficient smoothers with a high degree of parallelism. In combination with matrix-based multigrid methods on unstructured meshes our smoothers provide powerful solvers that are applicable across a wide range of parallel computing platforms and almost arbitrary geometries. We describe the configuration of our smoothers in the context of the portable *lmpLAtoolbox* and the *HiFlow*^{3} parallel finite element package. In our approach, a single source code can be used across diverse platforms including multicore CPUs and GPUs. Highly optimized implementations are hidden behind a unified user interface. Efficiency and scalability of our multigrid solvers are demonstrated by means of a comprehensive performance analysis on multicore CPUs and GPUs.

Keywords: Parallel smoothers, unstructured meshes, matrix-based multigrid, multi-coloring, power(q)-pattern method, FSAI, multi-core, GPUs

* Corresponding author: dimitar.lukarski@kit.edu

1 Introduction

The need for high accuracy and short simulation times relies both on efficient numerical schemes and appropriate scalable parallel implementations. The latter point is crucial in the context of fine-grained parallelism in the prospect of the manycore era. In particular, graphics processing units (GPUs) open new potentials in terms of computing power and internal bandwidth values. These architectures have a significant impact on the design and implementation of parallel algorithms in numerical simulation. The algorithms need to be laid out such that thousands of fine-grained threads can run in parallel in order to feed hundreds of computing units. Portability of solver concepts, codes and performance across several platforms is further a major issue in the era of highly capable multicore and accelerator platforms.

Multigrid methods rely on the decomposition of the error in low- and high-frequency contributions. The so-called smoothers damp out the high frequency contributions of the error at a given level. Adequate prolongation and restriction operators allow to address the considered problem on a hierarchy of discretizations and by this means cover the full spectral range of error contributions only on the basis of these smoothers (see [20] and references therein for further details). For full flexibility of the solvers in the context of complex geometries, stencil-based multigrid methods need to be replaced by more flexible concepts. We use the approach of matrix-based multigrid where all operations – i.e. smoothers, grid transfers and residual computation – are represented by sparse matrix-vector multiplications (SpMV). This approach is shown to work well on modern multicore platforms and GPUs [3]. Moreover, the restriction to basic algorithmic building blocks is the key technique for building portable solvers. The major challenge with respect to parallelism is related to the definition and implementation of adequate parallel smoothers.

Flexible multigrid methods on emerging multicore technologies are subject of recent research. Parallel multigrid smoothers and preconditioners on regularly structured tensor-product meshes (with a fixed number of neighbors but arbitrary displacements) are considered in [7]. The author discusses parallel implementation aspects for several platforms and shows integration of accelerators into a finite element software package. In [6], geometric multigrid on unstructured meshes for higher order finite element methods is investigated. A parallel Jacobi smoother and grid transfer operators are assembled into sparse matrix representation leading to efficient solvers on multicore CPUs and GPUs. In [5] performance results for multigrid solvers based on the *sparse approximate inverse* (SPAI) technique are presented. An alternative approach without the need for mesh hierarchies are algebraic multigrid methods (AMG) [10]. An implementation and performance results of an AMG on GPUs are discussed in [8].

In this work we propose a new parallel smoother based on the *power(q)-pattern enhanced multi-colored ILU(p) factorization* [14]. We compare it to multi-colored splitting-type smoothers and FSAI smoothers. These approaches provide efficient smoothers with scalable parallelism across multicore CPUs and GPUs. Various hardware platforms can be easily used in the context of these solvers by means of the portable implementation based on the *lmpLAtoolbox* in the parallel finite element software package *HiFlow*^{3} [2]. The capability of the proposed approach is demonstrated in a comprehensive performance analysis.

This paper is organized as follows. In Section 2 we describe the context of matrix-based multigrid methods. Section 3 describes the parallel concepts for efficient and scalable smoothers. Implementation aspects and the approach taken in the HiFlow^3 finite element method (FEM) package are presented in Section 4. The numerical test problem is the Poisson problem on an L-shaped domain. The impact of the choice and the configuration of the smoothers on the MG convergence is investigated in Section 5. A performance analysis with respect to solver times and parallel speedups on multicore CPUs and GPUs is the central theme in Section 6. After an outlook on future work in Section 7 we conclude in Section 8.

2 Matrix-based Multigrid Methods

Multigrid (MG) methods are usually used to solve large sparse linear systems of equations arising from finite element discretizations (or related techniques) of partial differential equations (PDEs) – typically of elliptic type. Due to the ability to achieve asymptotically optimal complexity, MG has been proven to be one of the most efficient solvers for this type of problems. In contrast to Krylov subspace solvers, the number of needed iterations in the MG method in order to achieve a prescribed accuracy does not depend on the number of unknowns N or the grid spacing h. Hence, the costs on finer grids only increase linearly in N due to the complexity of the applied operators. In this sense, MG is superior to other iterative methods (see e.g. [20, 9]).

The main idea of MG methods is based on coarse grid corrections and the smoothing properties of classical iterative schemes. High frequency error components can be eliminated efficiently by using elementary iterative solvers – such as Jacobi or Gauß-Seidel. On the other hand, smooth error components cannot be reduced efficiently on fine grids. This effect can be mitigated by performing a sequence of coarsening steps and transferring the smooth errors by restricting the defect to coarser mesh levels. On each coarser mesh, pre-smoothing is applied to reduce error components inherited by the grid transfer. Moreover, on this coarser level the smooth components can be damped much faster. When the coarsest level is reached, the so-called coarse grid problem has to be solved exactly or by an iterative method with sufficient accuracy. By applying the prolongation operator to the corrected defect the next finer level can be reached again. Any remaining high frequency error components can be eliminated by post-smoothing. This recursive execution of inter-grid transfer operators and smoothers is called the MG cycle. Details on the basic properties of the MG method as well as error analysis and programming remarks can be found in [20, 9, 18, 4] and in references provided therein. Aspects of parallel MG are e.g. discussed in [16].

The MG cycle is an iterative and recursive method, shown in Algorithm 1. Here, $L_h u_h = f_h$ is the discrete version of the underlying PDE on the refinement level with mesh size h. On each level, pre- and post-smoothing is applied in the form $u_h^{(n)} = \text{RELAX}^{\nu}(\tilde{u}_h^{(n)}, L_h, f_h)$. The parameter ν denotes the number of smoothing iterations. Grid transfer operators are denoted by I_h^H (restriction) and I_H^h (prolongation) where h is representing the fine grid size and H is the coarse grid size. The transferred residual r_H is the input to the MG iteration on the coarser level where the initial value for the error is taken by zero. The parameter γ specifies the number of two-grid cycle iterations on each level and thus specifies how often the coarsest level is visited. For $\gamma = 1$ we obtain so called V-cycles whereas $\gamma = 2$ results in W-cycles. The obtained solution after each cycle serves as the input for the successive cycle.

Algorithm 1: Multigrid cycle: $u_h^{(n)} = MG(u_h^{(n-1)}, L_h, f_h, \nu_1, \nu_2, \gamma)$

```
(1) $\bar{u}_h^{(n)} := \text{RELAX}^{\nu_1}(u_h^{(n-1)}, L_h, f_h)$          pre-smoothing
(2) $r_h^{(n)} := f_h - L_h \bar{u}_h^{(n)}$                             compute residual
(3) $r_H^{(n)} := I_h^H r_h^{(n)}$                                       restriction
(4) if $(H == h_0)$ then
        $L_H e_H^{(n)} = r_H^{(n)}$                                      exact solution on the coarse grid
(5) else
        $e_H^{(n)} = MG(0, L_H, r_H^{(n)}, \nu_1, \nu_2, \gamma)$        recursion
    end if
(6) $e_h^{(n)} := I_H^h e_H^{(n)}$                                       prolongation
(7) $\tilde{u}_h^{(n)} := \bar{u}_h^{(n)} + e_h^{(n)}$                   correction
(8) $u_h^{(n)} = \text{RELAX}^{\nu_2}(\tilde{u}_h^{(n)}, L_h, f_h)$      post-smoothing
```

In this work, we consider matrix-based geometric MG methods [19]. In contrast to stencil-based geometric MG methods, all differential operators and grid transfer operators are not expressed by fixed stencils on equidistant grids but have the full flexibility of sparse matrix representations. On the one hand, this approach gives us flexibility with respect to complex geometries, non-uniform grids resulting from local mesh refinements, and space-dependent coefficients in the underlying PDE. On the other hand, the solvers can be built upon standard building blocks of numerical libraries.

The convergence and robustness of MG methods strongly depend on the applied smoothers. While classical MG performs very well with additive smoothers such as Gauß-Seidel or Jacobi, strong anisotropies require more advanced smoothers in order to achieve full MG efficiency. In the latter case, incomplete LU decompositions have demonstrated convincing results, e.g. for 2D cases in [21].

3 Parallel Smoothers for Matrix-Based Multigrid

Stencil-based geometric MG methods can be efficiently performed in parallel by domain sub-structuring and halo exchange for the stencils [16]. In contrast, the parallel implementation of a matrix-based MG requires more work. While the grid transfer operators are explicit updates with sufficient locality, parallel smoothers rely on implicit steps where typically triangular systems need to be solved. And as we are working on unstructured meshes, straightforward parallelization strategies like wave-front or red-black-coloring cannot be applied. Therefore, the focus of our work is directed to fine-grained parallelism for the smoothing step. We consider additive and multiplicative matrix splittings used for designing efficient parallel smoothers considering the following two restrictions: first, the proposed smoothers should rely on a high degree of parallelism and should show significant speedups on multicore CPUs and GPUs. Second, the parallel versions of the smoothers should maintain smoothing properties comparable to their sequential version. The considered smoothers are based on the block-wise decomposition into small sub-matrices. In the additive and multiplicative splittings, typically a large amount of forward and backward sweeps in triangular solvers needs to be processed.

For the description of the proposed parallel smoothers we point out the link between smoothers and preconditioning techniques. Iterative solvers can generally be interpreted in fixed point form by $x_{k+1} = Gx_k + f$ where the linear system of the type $Ax = b$ is transformed by the additive splitting $A = M + N$ and taking $G = M^{-1}N = M^{-1}(M - A) = I - M^{-1}A$ and $f = M^{-1}b$. This version can be reformulated as a preconditioned defect correction scheme given by

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

In this context, we apply preconditioners M as smoothers for the linear system $Ax = b$. Additive preconditioners are standard splitting schemes typically based on the block-wise decomposition $A = D + L + R$ where L is a strictly lower triangular matrix, R is a strictly upper triangular matrix, and D is the matrix containing the diagonal blocks of A. We choose $M = D$ (Jacobi), $M = D + L$ (Gauß-Seidel) or $M = (D + L)D^{-1}(D + R)$ (symmetric Gauß-Seidel). For multiplicative splittings we choose $M = LU$ in (1) where L is a lower triangular and U is an upper triangular matrix. For incomplete LU (ILU) factorizations with or without fill-ins we decompose the system matrix A into the product $A = LU + R$ with a remainder matrix R that absorbs unwanted fill-in elements. Typically, diagonal entries of L are taken to be one and both matrices L and U are stored in the same data structure (omitting the ones). The quality of the approximation in this case depends on the number of fill-in elements. The third class of considered parallel smoothers are the approximate inverse preconditioners. Here, we are focusing on the *factorized sparse approximate inverse* (FSAI) algorithms [17] that compute a direct approximation of A^{-1}. These schemes are based on the minimization of the Frobenius norm $|I - GA|_F$ where one looks for a symmetric preconditioner in the form $G := G_L^T G_L$. In other words – one directly builds an approximation of the Cholesky decomposition based on a given sparse matrix structure. FSAI(1) uses the sparsity pattern of A, FSAI(q), $q \geq 2$, uses the sparsity pattern of $|A|^q$ respectively.

In order to harness parallelism within each block of the decomposition we apply multi-coloring techniques [14]. For splitting-based methods (like Gauß-Seidel and SOR) and ILU(0) decompositions without fill-ins the original sparsity pattern of the system matrix is preserved in the additive or multiplicative decompositions. Here, only subsets of the original sparsity pattern are populated and no additional matrix elements are inserted. Before applying the smoother, the matrix is re-organized such that diagonal blocks in the block decomposition are diagonal itself. Then, inversion of the diagonal blocks is just an easy vector operation [13].

Furthermore, we allow fill-ins (i.e. additional matrix elements) in the ILU(p) factorization for achieving a higher level of coupling with increased efficiency. The power(q)-pattern method is applied to ILU(p) factorizations with fill-ins [14]. This method is based on an incomplete factorization of the system matrix A subject to a predetermined non-zero pattern derived from a multi-coloring analysis of the matrix power $|A|^q$ and its associated multi-coloring permutation π. It has been proven in [14] that the obtained sparsity pattern is a superset of the modified ILU(p) factorization applied to $\pi A \pi^{-1}$. As a result, for $q = p + 1$ this modified ILU(p,q) scheme applied to the multi-colored system matrix has no fill-ins into its diagonal blocks. This leads to an inherently parallel execution of triangular ILU(p,q) sweeps and hence to a parallel and efficient smoother. The degree of parallelism can be increased by taking $q < p + 1$ at the expense of some fill-ins into the diagonal blocks. In this scenario (e.g. for the ILU(1,1) smoother in our experiments) we use a drop-off technique that erases fill-ins into the diagonal blocks. These techniques have already been successfully tested in the context of parallel preconditioners [14]. The major advantage is that multi-coloring can be applied before performing the ILU(p) with additional fill-ins – where fill-ins only occur outside the blocks on the diagonal. By precomputing the superset of the data distribution pattern we also eliminate costly insertion of new matrix elements into dynamic data structures and obtain compact loops [14].

4 Implementation Aspects

Flexibility of solution techniques and software is a decisive factor in the current computing landscape. We have incorporated the described multigrid solvers and parallel smoothers into the multi-platform and multi-purpose parallel finite element software package HiFlow^3 [11, 2]. With the concept of object-oriented programming in C++, HiFlow^3 provides a flexible, generic and modular framework for building efficient parallel solvers and preconditioners for PDEs of various types. HiFlow^3 tackles productivity and performance issues by its conceptual approach. It is structured in several modules. Its linear algebra operations are based on two communication and computation layers, the *LAtoolbox* for inter-node operations and the *lmpLAtoolbox* for intra-node operations in a parallel system. The lmpLAtoolbox has backends for multiple platforms with highly optimized implementations – hiding all hardware details from the user while maintaining optimal usage of resources. Numerical solvers in HiFlow^3 are built on the basis of unified interfaces to the different hardware backends and parallel programming approaches – where only a single code base is required for all platforms. By this means, HiFlow^3 is portable across diverse computing systems including multicore CPUs, CUDA-enabled GPUs and OpenCL-capable platforms. The modular approach also allows an easy extension to emerging systems with heterogeneous configuration. The main modules of HiFlow^3 are described in the following: Mesh, DoF/FEM and Linear Algebra. See Figure 1 for an abstraction.

Mesh - This module provides a set of classes and functions for handling general types of meshes. This library deals with complex unstructured distributed meshes with different types of cells via a uniform interface. The data structures provided can be used to represent meshes of arbitrary topological and geometrical dimension, and even with different cell types in one and the same mesh. It is possible to create hierarchies of meshes through refinement and also to coarsen cells given such a hierarchy.

DoF/FEM (Degrees of Freedom / Finite Element Methods) - Treatment of the FEM and DoF is highly interdependent and therefore, the major idea of this module is to capture both sub-modules and unify them within a single scope. The DoF sub-module deals with the numbering of all DoF resulting from the finite element ansatz on each mesh cell. It can handle discontinuous and continuous finite element ansantz functions. This data is passed to the FEM sub-module. This part of the library handles the task of representing the chosen finite element ansatz on the reference cell and transforming it into the physical space depending on its geometry.

Linear Algebra - This module handles the basic linear algebra operations and offers (non-)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 to several platforms by providing different platform backends, e.g. to multicore CPUs (OpenMP, MKL) and GPUs (CUDA, OpenCL).

In this work we are only focusing on single node parallelism. Therefore, we do not consider the global MPI level. Based on the matrix and vector classes HiFlow^3 offers several Krylov subspace solvers and local additive and multiplicative preconditioners. Further information about this library, solvers and preconditioners can be found in [14, 12, 13, 15].

Implementation of Matrix-based Multigrid in HiFlow^3

The proposed multigrid solvers and smoothers have been implemented in the context of HiFlow^3. Starting on an initial mesh, the HiFlow^3 mesh module refines

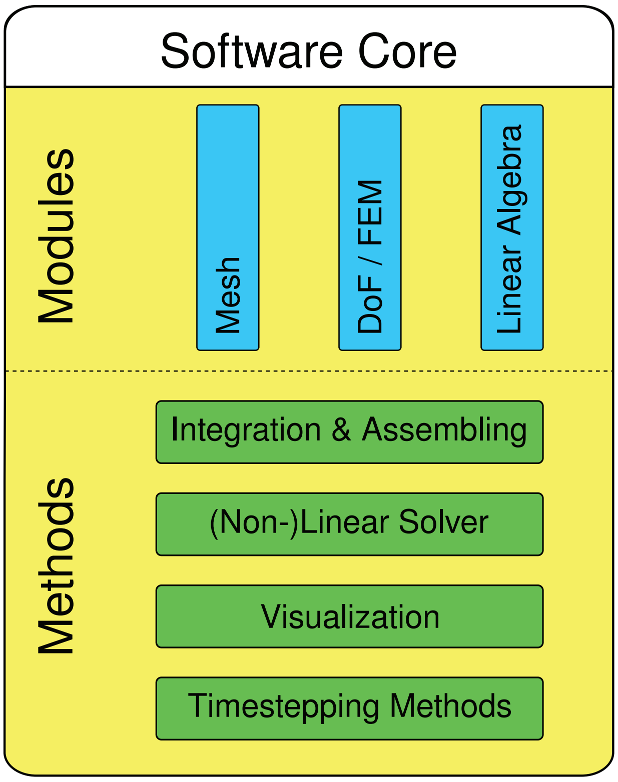

Fig. 1. Modular configuration of the parallel FEM package HiFlow^3.

the grid in order to reach the requested number of cells according to the demands for accuracy. The current version supports quadrilateral elements as well as triangles for two dimensional problems. In the three-dimensional case tetrahedrons and hexahedrons are used. For our two-dimensional test case, an unstructured mixed mesh containing both types of elements with local refinements has been used. On each refinement level, the corresponding stiffness matrix and the right hand side are assembled for the given equation. Our implementation performs a bi-linear interpolation to ascend to a finer level and a full weighting restriction to descend to a coarser level respectively. With the relation $I_{2h}^{h} = 2^{\dim}(I_{h}^{2h})^{T}$ for the intergrid transfer operators for the standard coarsening procedure, it is sufficient to assemble the prolongation matrix only.

Due to the modular approach of the lmpLAtoolbox, the matrix-based MG solver is based on a single source code for all different platforms (CPUs, GPUs, etc). The grid transformations and the defect computation are based on matrix-vector and vector-vector routines which can be efficiently performed in parallel on the considered devices. Moreover, a similar approach is taken for the smoothing steps based on the preconditioned defect correction (1). In this context, our smoothers make use of parallel preconditioners of additive and multiplicative type. See also [14] for a detailed description of the parallel concepts for preconditioners, the *power(q)-pattern enhanced multi-colored ILU(p) schemes* and a comprehensive performance analysis with respect to solver acceleration and parallel speedups on multicore CPUs and GPUs.

5 Numerical Experiments and Smoother Efficiency

As a numerical test problem we are solving the Poisson problem with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega, \\ u &= 0 \text{ on } \partial\Omega \end{aligned} \tag{2}$$

in the two-dimensional L-shaped domain $\Omega := (0,1)^2 \setminus (0,0.5)^2$ with a reentrant corner. For our test case we choose the right hand side $f = 8\pi^2 \cos(2\pi x)\cos(2\pi y)$. We solve (2) by means of bilinear Q1 elements on quadrilaterals and linear P1 elements on triangles [11, 2]. A uniform discretization of the L-shaped domain is depicted in Figure 2 (left). The numerical solution of (2) with boundary layers is detailed in Figure 2 (right). Due to the steep gradients at the reentrant corner we

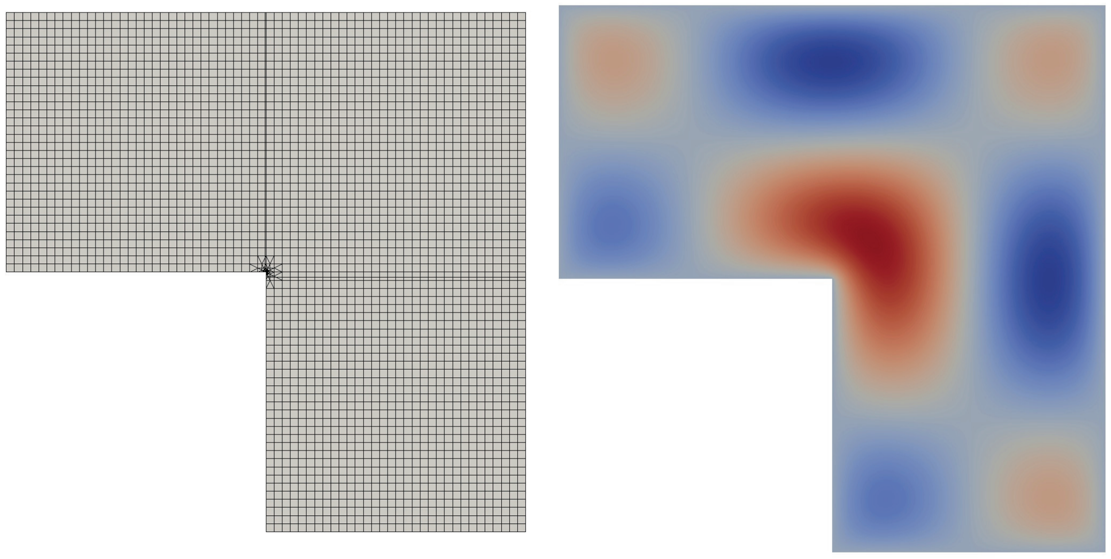

Fig. 2. Discretization of locally refined L-shaped domain (left) and discrete solution of the Poisson problem (2) (right).

are using a locally refined mesh in order to obtain a proper problem resolution. Figure 3 (left) shows a zoom-in into a uniformly refined coarse mesh. In the mesh in Figure 3 (right) we are using additional triangular and quadrilateral elements for local refinement and avoiding hanging nodes and deformed elements [1].

Our coarsest mesh in the multigrid hierarchy is a locally refined mesh based on triangular and quadrilateral cells with Q1 and P1 elements summing up to 3,266 degrees of freedom (DoF). By applying six global refinement steps to this coarse mesh we obtain the hierarchy of nested grids where the finest mesh has 3,211,425 DoF. Details of the mesh hierarchy are listed in Table 1. None of our grids is a uniform grid with equidistant mesh spacings. For the inter-grid operations we are using full weighting restrictions and bi-linear interpolations.

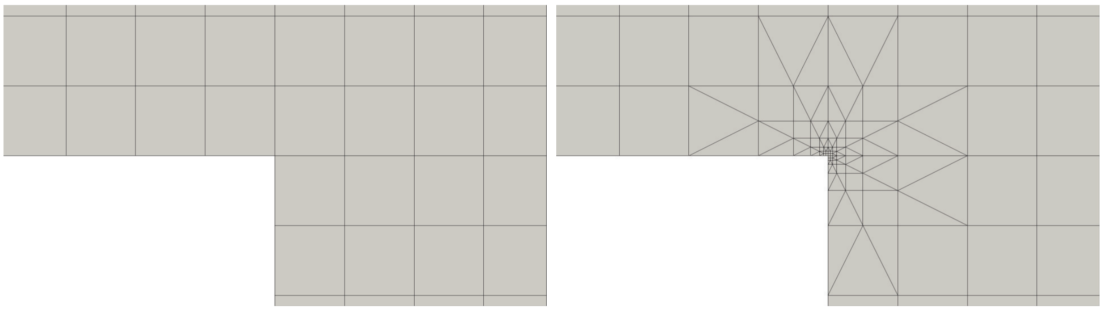

Fig. 3. Zoom-in to a uniform mesh with 3,201 DoF (left) and a locally refined mesh with 3,266 DoF (right) of the L-shaped domain around the reentrant corner – our coarsest MG mesh with level 1.

level	#cells	#DoF	$h_{\max}$	$h_{\min}$ at $(0.5, 0.5)$
1	3,177	3,266	0.015625	0.000488
2	12,708	12,795	0.0078125	0.000244
3	50,832	50,645	0.00390625	0.000122
4	203,328	201,513	0.001953125	0.0000610
5	813,312	803,921	0.000976563	0.0000305
6	3,253,248	3,211,425	0.000488281	0.00001525

Table 1. Characteristics of the six refinement levels of the MG hierarchy.

For the motivation of our locally refined grids we consider the following example. The gradient of the solution on a fine uniform mesh with 3,149,825 DoF shows a low resolution as can be seen in Figure 4 (left). The gradient of the solution on the locally refined mesh with 3,211,425 DoF (our finest grid in the multigrid hierarchy) is approximated much more accurately as depicted in Figure 4 (right). Clearly, with the same amount of unknowns we can solve the problem with higher accuracy only on locally refined meshes. There is a significant improvement of approximation quality with less than 2 % additional grid points.

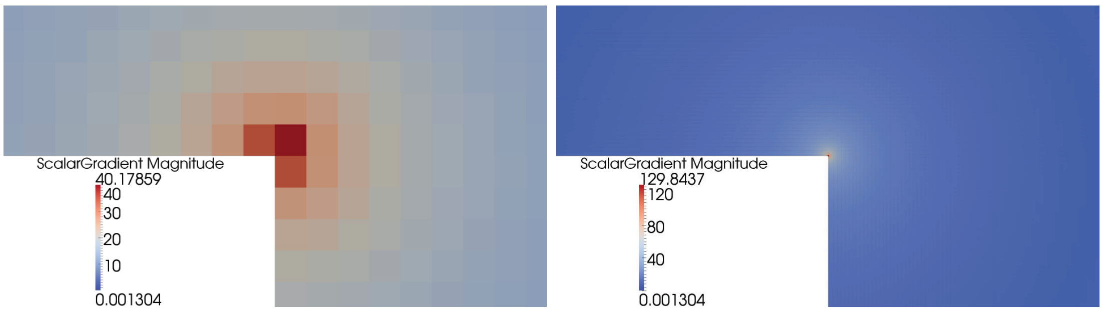

Fig. 4. Zoom-in plots of the gradient of the solution of (2)on a uniform mesh with 3,149,825 DoF (left) and on a locally refined mesh with 3,211,425 DoF (right) – our finest MG mesh with level 6.

In this work we study the behavior of parallel smoothers based on additive splittings (Jacobi, multi-colored Gauß-Seidel and multi-colored symmetric Gauß-

Seidel), incomplete factorizations (multi-colored ILU(0) and power(q)-pattern enhanced multi-colored ILU(p) with fill-ins) and FSAI smoothers. We investigate their properties with respect to high-frequency error reduction and convergence behavior of the V- and W-cycle parallel MG. In particular, we consider the multigrid behavior on different test platforms. Our numerical experiments are performed on a hybrid platform, a dual-socket Intel Xeon (E5450) quad-core system (with 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. We are only using double precision for the computations.

We perform several tests with different configurations for the pre- and post-smoothing steps. We determine the number of cycles required to achieve a relative residual less than 10^{-6}. In Table 2 the iteration counts are shown for the V-cycle based MG. Note, that the iteration counts in this table do not reflect the total amount of work. Some smoothing steps, e.g. ILU(p), need more work than others. From a theoretical point of view, a Gauß-Seidel smoothing step is half as costly as a symmetric Gauß-Seidel step; ILU(0) is cheaper than ILU(1,1) which is cheaper than ILU(1,2). For the MG solver, $\nu_1 + \nu_2$ is the number of smoothing steps on each level and should be kept low in order to reduce the total amount of work (and hence the solver time). For estimation of the corresponding work load, computing times are included for the MG solver on the GPU platform.

For Jacobi smoothers there is no convergence of the MG solver within 99 iterations. Multi-colored symmetric Gauß-Seidel (SGS) is better than multi-colored Gauß-Seidel (GS) in terms of iteration count. Multi-colored ILU smoothers are better than additive factorizations (GS, SGS). The quality of the ILU(p) schemes in terms of iteration counts gets better with increasing p. The drop-off technique for ILU(1,1) is providing only little improvements compared to ILU(0). Iteration counts for the FSAI smoothers are in the same range as the ILU smoothers. Best candidates with respect to reduced iteration counts are ILU(1,2) and FSAI(2). Minima with respect to the iteration count can be found for configurations where $\nu_1 + \nu_2 = 3$ or 4. For larger values there are no more significant improvements.

The run time results show that the benefits for the SGS smoother in terms of smoothing properties and reduced iteration count are eaten up in terms of run time due to the additional overhead in each smoothing step and an additional V-cycle. For the ILU smoothers the additional work complexity still yields improvements in run time. The best performance is obtained for the ILU(1,2) smoother. The drop-off technique for ILU(1,1) has some diminishment in performance. The FSAI smoothers give no particular improvements in run time compared to the other smoothers.

Table 3 shows iteration counts and run times for the W-cycle based MG solver. The iteration counts are reduced compared to the V-cycle based MG solver. However, the run times show that the W-cycle based MG solver is by a factor of 2 to 3 slower than the V-cycle based MG solver. The W-cycle based MG solver is slower because more smoothing steps are performed on coarser grids. This involves a larger number of calls to SpMV routines for small matrices with

(ν_1, ν_2)	(0,1)	(0,2)	(0,3)	(0,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(2,0)	(2,1)	(2,2)
Jacobi [#its]	>99	>99	>99	>99	>99	>99	>99	>99	>99	>99	>99	>99
GS [#its]	30	14	10	8	35	16	10	8	7	20	12	9
time [sec]	5.75	3.89	3.62	3.58	6.44	4.43	3.63	3.59	3.73	5.33	4.34	4.03
SGS [#its]	23	13	9	8	28	14	10	8	7	17	11	9
time[sec]	4.89	4.17	3.88	4.28	5.71	4.49	4.28	4.28	4.51	5.29	4.69	4.80
ILU(0) [#its]	18	11	8	6	25	11	8	7	6	15	8	7
time [sec]	3.80	3.49	3.40	3.19	5.00	3.50	3.40	3.72	3.82	4.59	3.42	3.72
ILU(1,1) [#its]	18	10	7	6	23	11	8	7	6	14	8	7
time [sec]	4.30	3.73	3.56	3.85	5.26	4.10	4.06	4.49	4.66	5.06	4.06	4.49
ILU(1,2) [#its]	10	6	5	5	12	7	5	5	5	9	6	5
time [sec]	2.93	2.75	3.08	3.86	3.36	3.20	3.10	3.87	4.65	3.98	3.69	3.87
FSAI(1) [#its]	17	9	7	6	22	10	7	6	6	14	8	7
time [sec]	4.14	3.35	3.46	3.69	5.41	3.72	3.47	3.70	4.39	5.18	3.94	4.27
FSAI(2) [#its]	14	7	6	5	15	8	6	5	5	10	6	6
time [sec]	4.54	4.08	4.96	5.36	5.71	4.64	4.96	5.38	6.59	6.30	4.96	6.42
FSAI(3) [#its]	9	6	5	5	12	6	5	5	5	8	6	5
time [sec]	4.04	5.15	6.40	8.42	6.58	5.20	6.39	8.45	10.50	7.77	7.63	8.44

(ν_1, ν_2)	(2,3)	(2,4)	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)
Jacobi [#its]	>99	>99	>99	>99	>99	>99	>99	>99	>99	>99	>99	>99
GS [#its]	7	7	15	10	8	7	6	12	9	8	7	7
time [sec]	3.74	4.31	5.27	4.45	4.25	4.34	4.21	5.24	4.76	4.91	4.90	4.70
SGS [#its]	7	7	13	9	8	7	6	11	9	7	7	6
time [sec]	4.49	5.22	5.43	4.82	5.12	5.23	5.12	5.78	5.76	5.23	5.96	5.75
ILU(0) [#its]	6	6	11	7	6	6	6	9	7	6	6	5
time [sec]	3.83	4.45	4.54	3.72	3.82	4.44	5.01	4.68	4.44	4.44	5.06	4.75
ILU(1,1) [#its]	6	5	10	7	6	6	5	9	6	6	5	5
time[sec]	4.65	4.54	4.96	4.49	4.65	5.43	5.20	5.66	4.65	5.43	5.19	5.87
ILU(1,2) [#its]	5	5	8	6	5	5	5	7	6	5	5	5
time [sec]	4.66	5.43	4.79	4.64	4.65	5.43	6.21	5.30	5.58	5.43	6.22	6.99
FSAI(1) [#its]	6	6	11	8	6	6	6	10	7	6	6	6
time [sec]	4.39	5.07	5.40	4.86	4.39	5.08	5.77	6.09	5.08	5.08	5.80	6.48
FSAI(2) [#its]	5	5	8	6	6	5	5	8	6	5	5	5
time [sec]	6.60	7.82	7.05	6.43	7.88	7.80	9.02	8.97	7.87	7.79	9.03	10.23
FSAI(3) [#its]	5	5	8	6	5	5	5	7	5	5	5	4
time [sec]	10.49	12.58	11.06	10.08	10.48	12.56	14.62	12.57	10.49	12.56	14.60	13.38

Table 2. Number of MG V-cycles for different smoothers and total run times of the V-cycle based MG solver on the GPU for different smoother configurations; ν_1 is the number of pre-smoothing steps, ν_2 is the post-smoothing step count.

significant overhead (in particular kernel call overheads on the GPU). The best results for the W-cycle based MG are obtained for the ILU(0) smoother with Gauß-Seidel following next. Except of the FSAI smoothers, all other smoothers are on the same performance level.

(ν_1, ν_2)	(0,1)	(0,2)	(0,3)	(0,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(2,0)	(2,1)	(2,2)
GS [#its]	25	13	9	7	28	13	9	7	6	15	10	8
time [sec]	14.76	9.71	8.06	7.34	15.06	9.21	7.73	7.15	7.05	10.15	8.35	7.93
SGS [#its]	23	12	9	7	25	13	9	7	6	14	9	7
time [sec]	14.60	10.05	9.15	8.45	14.52	10.22	8.86	8.27	8.20	10.77	8.88	8.27
ILU(0) [#its]	19	11	8	6	21	11	8	6	5	12	8	6
time [sec]	11.33	8.61	7.80	7.09	11.31	8.32	7.61	7.02	6.85	8.79	7.58	7.00
ILU(1,1) [#its]	18	10	8	6	20	11	8	6	6	12	8	7
time [sec]	11.73	8.81	8.69	8.02	12.34	9.35	8.70	8.01	9.19	9.87	8.67	9.05
ILU(1,2) [#its]	10	6	5	4	11	7	5	4	4	7	5	4
time [sec]	10.30	8.34	8.43	8.17	10.80	9.33	8.40	8.18	9.23	9.17	8.39	8.15
FSAI(1) [#its]	18	9	6	5	19	10	7	5	5	10	7	6
time [sec]	20.32	13.05	10.30	9.59	20.43	13.41	11.08	9.39	10.10	13.52	11.05	10.57
FSAI(2) [#its]	14	7	5	4	15	8	5	4	4	8	6	5
time [sec]	19.03	12.83	11.26	10.68	20.64	14.00	11.23	10.64	11.81	14.33	12.71	12.55
FSAI(3) [#its]	9	5	4	4	10	6	4	4	4	6	5	4
time [sec]	14.64	11.73	11.95	14.15	16.95	13.37	11.82	14.01	16.02	14.05	14.00	13.98

(ν_1, ν_2)	(2,3)	(2,4)	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)
GS [#its]	6	6	10	8	7	6	5	9	7	6	5	5
time [sec]	7.01	7.77	8.33	7.91	7.94	7.75	7.25	8.51	7.94	7.75	7.22	7.87
SGS [#its]	6	6	10	8	7	6	5	8	7	6	6	5
time [sec]	8.20	9.16	9.60	9.22	9.35	9.18	8.67	9.22	9.34	9.16	10.14	9.50
ILU(0) [#its]	5	5	8	7	6	5	4	7	6	5	5	4
time [sec]	6.84	7.65	7.54	7.89	7.93	7.65	6.96	7.82	7.93	7.62	8.43	7.61
ILU(1,1) [#its]	6	5	9	7	6	5	5	7	6	5	5	4
time [sec]	9.16	8.88	9.37	9.04	9.14	8.87	9.86	8.97	9.12	8.86	9.86	8.94
ILU(1,2) [#its]	4	4	6	5	4	4	4	5	4	4	4	4
time [sec]	9.23	10.39	9.53	9.74	9.20	10.35	11.46	9.66	9.18	10.32	11.45	12.62
FSAI(1) [#its]	5	5	8	6	5	5	4	7	5	5	5	4
time [sec]	9.98	10.79	12.08	10.57	10.05	10.83	9.83	11.76	10.03	10.81	11.58	10.48
FSAI(2) [#its]	4	4	6	5	4	4	4	6	4	4	4	4
time [sec]	11.86	13.25	13.08	12.60	11.84	13.21	14.46	14.48	11.80	13.18	14.47	15.81
FSAI(3) [#its]	4	4	5	4	4	4	3	5	4	4	4	3
time [sec]	16.41	18.82	14.68	13.97	16.40	18.78	16.37	17.40	16.33	18.72	21.12	18.06

Table 3. Number of multigrid W-cycles for different smoothers and total run times of the W-cycle based MG solver on the GPU for different smoother configurations; ν_1 is the number of pre-smoothing steps, ν_2 is the post-smoothing step count.

In Table 4 the run times and iteration counts for the best V-cycle and W-cycle based smoother configurations for the parallel MG solvers on the GPU are summarized. The first column in this table specifies the optimal values for the parameters ν_1 and ν_2 that correspond to the number of pre- and post-smoothing steps on each refinement level. The third column gives the number of necessary MG cycle iterations.

	V-cycle based MG			W-cycle based MG		
Smoother	(ν_1, ν_2)	time [sec]	#its	(ν_1, ν_2)	time [sec]	#its
GS	(0,4)	3.58	8	(2,3)	7.01	6
SGS	(0,3)	3.88	9	(2,3)	8.20	6
ILU(0)	(0,4)	3.19	6	(2,3)	6.84	5
ILU(1,1)	(0,3)	3.56	7	(1,3)	8.01	6
ILU(1,2)	(0,2)	2.75	6	(2,2)	8.15	4
FSAI(1)	(0,2)	3.35	9	(1,3)	9.39	5
FSAI(2)	(0,2)	4.08	7	(1,3)	10.64	4
FSAI(3)	(0,1)	4.04	9	(0,2)	11.73	5

Table 4. Minimal run times and corresponding iteration counts for the V-cycle and W-cycle based parallel MG solvers on the GPU for various smoothers and optimal configurations of the corresponding number of pre- and post-smoothing steps (ν_1, ν_2).

Compared to the results of the Krylov subspace solvers, MG performance is superior. In Table 5 the run times are listed for the preconditioned *conjugate gradient* (CG) method on the CPU and GPU test platforms. The considered smoothers are used as preconditioners in this context. The number of CG iterations is given in the first row. We find that all preconditioners significantly reduce the number of iterations. Run times are presented for the sequential CPU version, the eight-core OpenMP parallel version, and the GPU version. We see that the MG solver on the GPU is faster by a factor of up to 60 than the preconditioned CG solver on the GPU. The best preconditioner is ILU(1,2) on the GPU.

Precond	None	Jacobi	SGS	ILU(0)	ILU(1,1)	ILU(1,2)	FSAI(1)	FSAI(2)	FSAI(3)
# iter	5,650	4,167	2,323	2,451	2,066	1,387	2,198	1,493	1,139
Sequential	1492s	1134s	1433s	1472s	1347s	1498s	1271s	950.4s	1082.5s
OpenMP	604.4s	573.4s	662.6s	676.9s	630.0s	589.7s	435.5s	438.5s	500.302s
GPU	273s	218.3s	186.6s	195.1s	206.7s	158.0s	204.1s	280.0s	359.662s

Table 5. Run times in seconds and iteration counts for the preconditioned CG solver for various preconditioners on the CPU (sequential and eight-core OpenMP parallel) and on the GPU.

In the following we consider the smoothing properties in more details. In Figure 5 the initial error with its high oscillatory parts is shown for a random initial guess. In the following figures we present the reduction of the error (compared to the final MG solution) after one and three smoothing steps with the corresponding smoother. For this experiment we choose the locally refined mesh of level 2 with 12,795 DoF (the second coarsest mesh in our MG hierarchy). We see that the effect from Jacobi smoothing as shown in Figure 6 is worse than that from Gauß-Seidel smoothing shown in Figure 7 which itself is worse than the symmetric Gauß-Seidel smoother presented in Figure 8. Even more smooth results are observed for ILU(0), ILU(1,1) and ILU(1,2) as shown in Figure 9, 10 and 11. With the FSAI smoothers, some higher order oscillations are still observed after the initial smoothing step as can bee seen in Figures 12, 13 and 14.

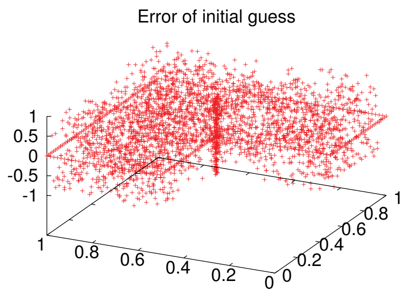

Fig. 5. Initial error for the MG iteration with random initial guess.

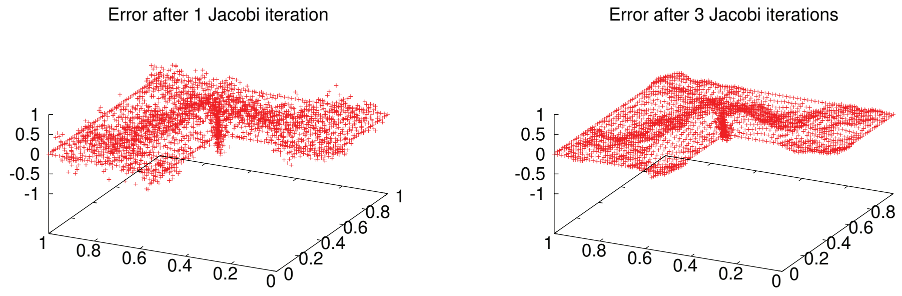

Fig. 6. Damped error after 1 (left) and 3 (right) Jacobi smoothing steps.

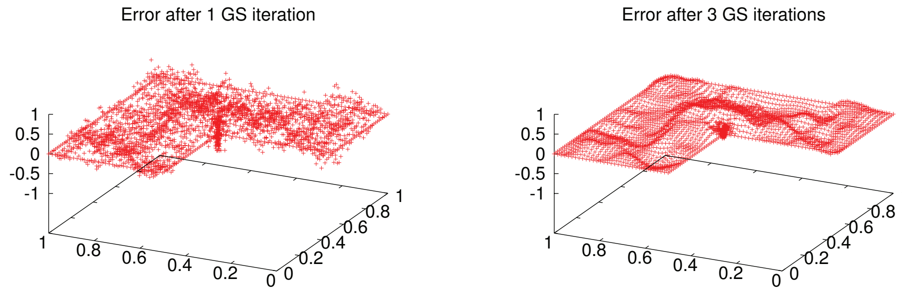

Fig. 7. Damped error after 1 (left) and 3 (right) Gauß-Seidel smoothing steps.

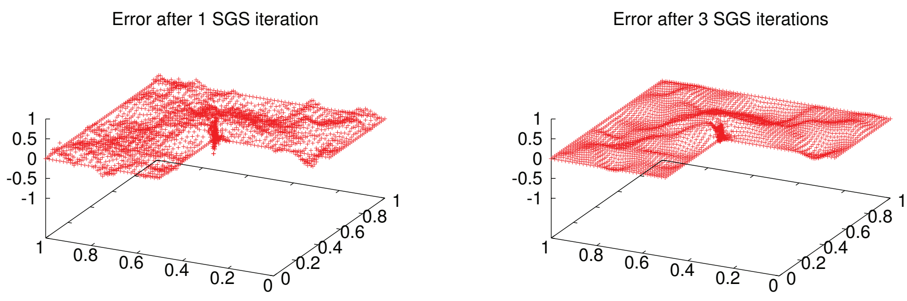

Fig. 8. Damped error after 1 (left) and 3 (right) symmetric Gauß-Seidel smoothing steps.

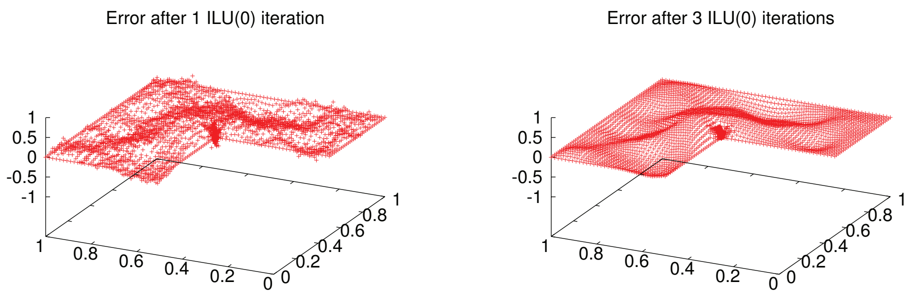

Fig. 9. Damped error after 1 (left) and 3 (right) ILU(0) smoothing steps.

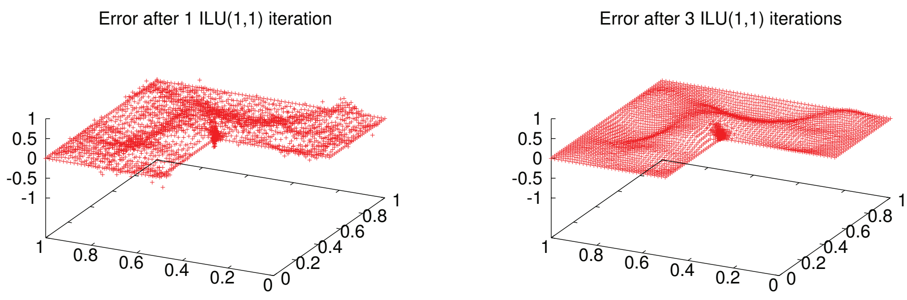

Fig. 10. Damped error after 1 (left) and 3 (right) ILU(1,1) smoothing steps.

6 Performance Analysis on Multicore CPUs and GPUs

In this section we conduct a performance analysis with respect to the different test platforms. We assess the corresponding solver times and the parallel speedups. In Figure 15 we compare run times of the V-cycle (left) and W-cycle (right) MG solver with various smoothers for the Poisson problem on the L-shaped domain. It shows the results for the sequential version and the OpenMP parallel version on eight cores of the CPU as well as for the GPU version. For this test problem, there are no significant differences in performance for the tested smoothers on a specific platform. The multiplicative ILU-type smoothers

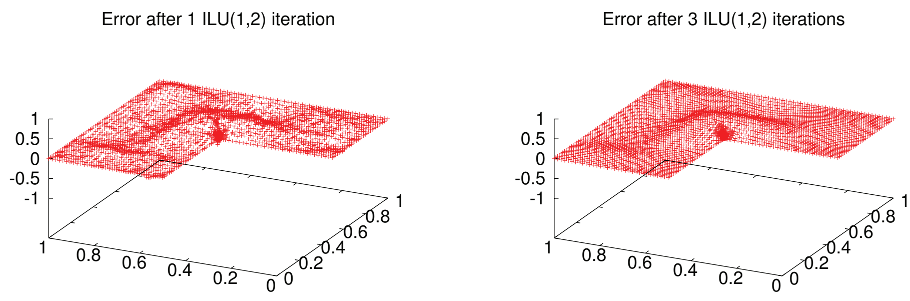

Fig. 11. Damped error after 1 (left) and 3 (right) ILU(1,2) smoothing steps.

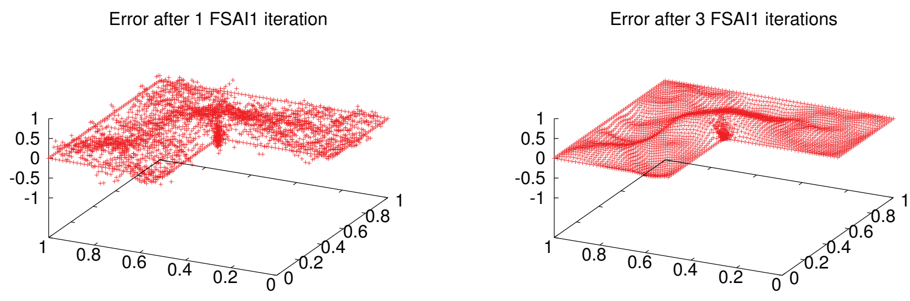

Fig. 12. Damped error after 1 (left) and 3 (right) FSAI(1) smoothing steps

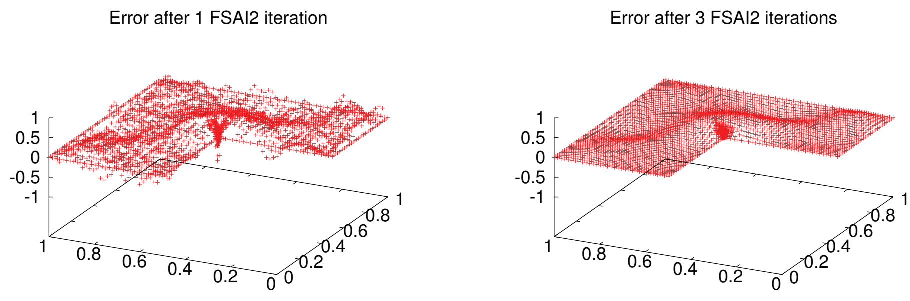

Fig. 13. Damped error after 1 (left) and 3 (right) FSAI(2) smoothing steps

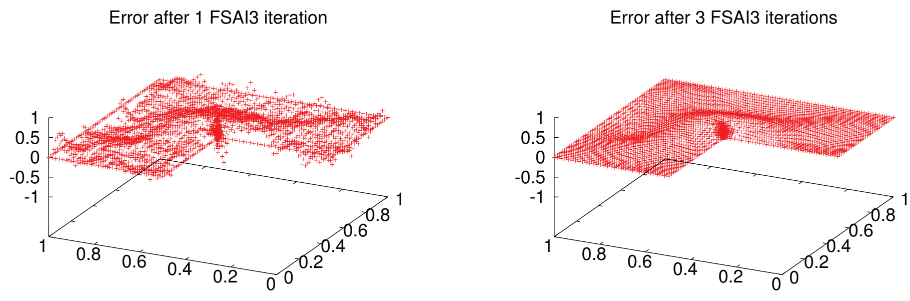

Fig. 14. Damped error after 1 (left) and 3 (right) FSAI(3) smoothing steps

are slightly faster than the additive splitting-type smoothers. The ILU-based smoothers are more efficient in terms of smoothing properties and reduced iteration counts, but they are more expensive to be executed. In total, both effects interact such that the total execution time is only slightly better. Best performance results are obtained for the ILU(1,2) smoother based MG solver on the GPU. In this 2D Poisson test problem with unstructured grids based on Q1 and P1 elements, the resulting stiffness matrix has only six colors in the multicoloring decomposition. By increasing the sparsity pattern with respect to the structure of $|A|^2$ used for building the ILU(1,2) decomposition, we obtain 16 colors. In this case the triangular sweeps can still be performed with a high degree of parallelism. However, on the CPU the FSAI algorithms perform better due to utilization of cache effects. The FSAI algorithms are performed by relying on parallel matrix-vector multiplications only. This is in contrast to the multicoloring technique which re-orders the matrix by grouping the unknowns. This distribution is performing better on the GPU due to the lack of bank conflicts in the sparse matrix-vector multiplications. On the other hand, the FSAI algorithms are performing better on the cache-based architecture due to the large number of elements that can benefit from data reuse.

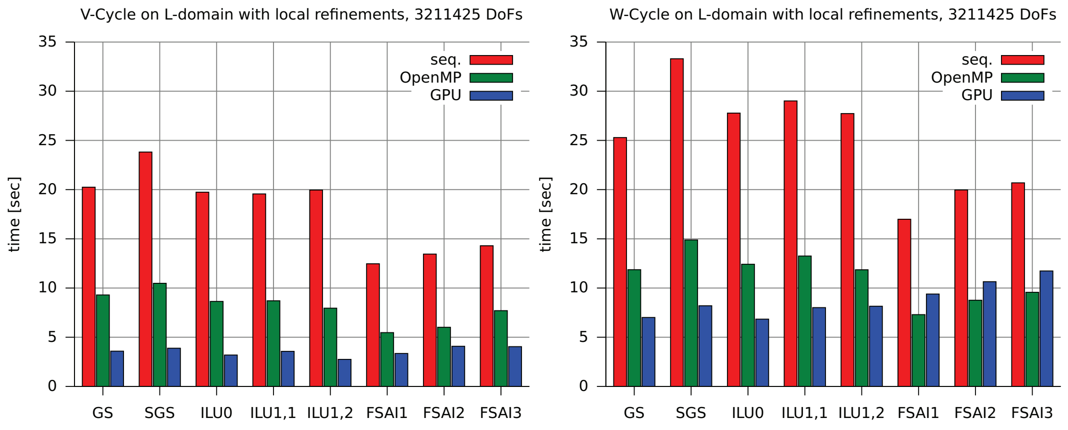

Fig. 15. Run times of V-cycle (left) and W-cyle (right) MG solver with various smoothers for the Poisson problem on the L-shaped domain: sequential version and OpenMP parallel version on eight cores on the CPU and GPU version.

The parallel speedups of the V-cycle and W-cycle based MG solvers are detailed in Figure 16. The OpenMP parallel speedup is slightly above two. In the sequential run on a single CPU core, about less than one third of the eight core peak bandwidth can be utilized (see measurements in [13]). Therefore, the speedup of the eight-core OpenMP parallel version is technically limited by a factor of less than three on this particular test platform. This performance expectations are reflected by our measurements reported here. The GPU version is by a factor of two to three faster than the OpenMP parallel version. These factors are in good conformance with experience for GPU acceleration of bandwidth-bound kernels. For the FSAI smoothers, the speedup on the GPU falls a little bit behind since it is better suited for CPU architectures. The presented parallel speedup results demonstrate the scalability of our parallel approach for efficient multigrid smoothers. Parallel multigrid solvers are typically affected by bad communication/computation ratios and load imbalance on coarser grids. As we are working on shared memory type devices, these influences do not occur. In contrast however, our matrix-based multigrid solver depends by construction on the calls to SpMV kernels. On coarser grids, these kernels have a significant call overhead at the expense of parallel efficiency. This has a direct impact on the speedups of the W-cycle based MG solvers on the GPU – as can be seen in Figure 16 (right).

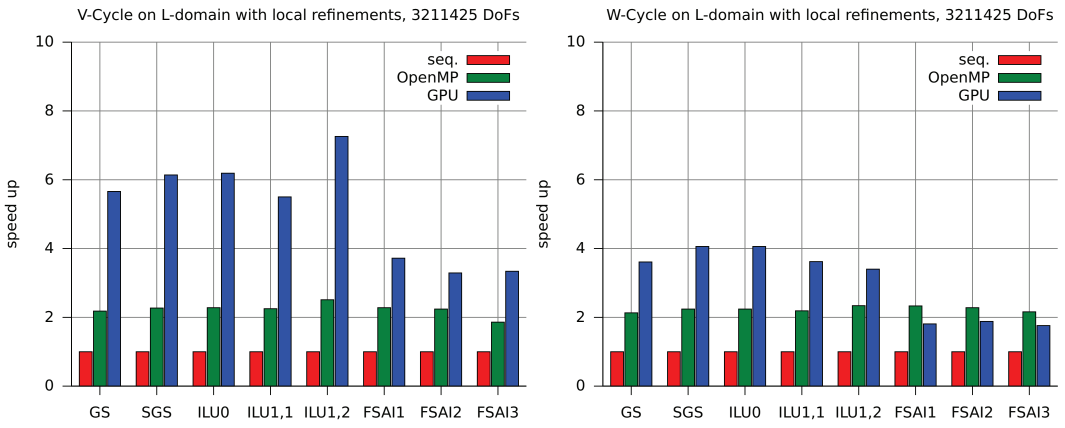

Fig. 16. Parallel speedup of the V-cycle (left) and W-cycle (right) based multigrid solvers for various smoothers.

Similar results are obtained for the performance numbers and speedups of the preconditioned CG solver. In the left part of Figure 17 run time results are listed for various preconditioners. The right figure details corresponding speedups for the OpenMP parallel version and the GPU implementation. Speedups for the CG are larger than those for the MG solver. Although both solvers consist of the same building blocks, CG is more efficient since it fully relies on the finest grid of the MG hierarchy with huge sparse matrices. In contrast, the MG solver is doing work on coarser grids where call overheads for SpMV kernels have a significant influence on the results.

7 Outlook on Future Work

In our future work the proposed multigrid solvers will be extended with respect to higher order finite element methods. This is basically a question of inter-grid transfer operators since HiFlow^3 allows finite elements of arbitrary degree. Further performance results will be included for our OpenCL backends. More importantly our solvers, smoothers and preconditioners will be extended for parallelism on distributed memory systems. The LAtoolbox in HiFlow^3 is built on an MPI-communication and computation layer. The major work has

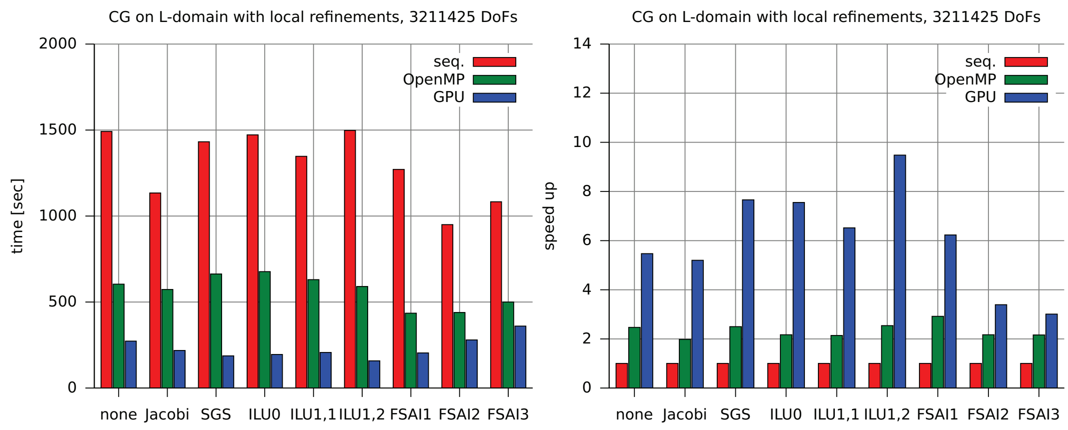

Fig. 17. Run times and parallel speedups for the preconditioned CG solvers on the CPU (sequential and eight-core OpenMP parallel version) and on the GPU.

to be done on the algorithmic side. Intensive research needs to be invested into hybrid parallelization where MPI-based parallelism is coupled with node-level parallelism (OpenMP, CUDA). MPI-parallel versions of efficient preconditioners and smoothers on unstructured meshes are subject of current research. Moreover, the hierarchical memory sub-systems and hybrid configurations of modern multi- and manycore systems require new hardware-aware algorithmic approaches. Due to the limited efficiency of SpMV kernels on GPUs with respect to coarse grids, U-cycles for the MG solvers should be investigated that stop on finer grids.

8 Conclusion

Matrix-based multi-grid solvers on unstructured meshes are efficient numerical schemes for solving complex and highly relevant problems. The paradigm shift towards manycore devices brings up new challenges in two dimensions: first, the algorithms need to express fine-grained parallelism and need to be designed with respect to scalability to hundreds and thousands of cores. Secondly, software solutions and implementations need to be designed flexible and portable in order to exploit the potential of various platforms in a unified approach. The proposed techniques for parallel smoothers and preconditioners provide both efficient and scalable parallel schemes. We have demonstrated how sophisticated mathematical techniques can be extended with respect to scalable parallelism and how these techniques can harness the computing power of modern manycore platforms. In particular, with the formulation in terms of SpMV kernels the capabilities of GPUs can be exploited. With the described solvers, solution times for realistic problems can be kept at a moderate level. And with the concept of our generic and portable software package, the numerical solvers can be used on a variety of platforms on a single code base. The users are freed from specific hardware knowledge and platform-oriented optimizations. We have reported speedups and we have demonstrated that even complex algorithms can be successfully ported to GPUs with additional performance gains. The proposed ILU(1,2) smoother and the FSAI(2) smoother show convincing performance and scalability results for parallel matrix-based MG. The proposed V-cycle based MG solvers with parallel smoothers on the GPU are by a factor of 60 faster than the preconditioned CG solvers on the GPU. But due to the absence of coarse grid operations, the CG solvers have slightly better scalability properties on GPUs.

Acknowledgements

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

References

1. Andrew, R.B., Sherman, A.H., Weiser, A.: Some refinement algorithms and data structures for regular local mesh refinement (1983)
2. 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. Tech. Rep. 2010-06, EMCL, KIT (2010). URL http://www.emcl.kit.edu/preprints/emcl-preprint-2010-06.pdf
3. 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. ACM, New York (2009). DOI http://doi.acm.org/10.1145/1654059.1654078
4. Demmel, J.W.: Applied numerical linear algebra. SIAM, Philadelphia (1997)
5. Geveler, M., Ribbrock, D., Göddeke, D., Zajac, P., Turek, S.: Efficient finite element geometric multigrid solvers for unstructured grids on GPUs. In: P. Iványi, B.H. Topping (eds.) Second Int. Conf. on Parallel, Distributed, Grid and Cloud Computing for Engineering, p. 22 (2011). DOI 10.4203/ccp.95.22
6. Geveler, M., Ribbrock, D., Göddeke, D., Zajac, P., Turek, S.: Towards a complete FEM-based simulation toolkit on GPUs: Geometric multigrid solvers. In: 23rd Int. Conf. on Parallel Computational Fluid Dynamics (ParCFD'11) (2011)
7. Göddeke, D.: Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters. Ph.D. thesis, Technische Universität Dortmund (2010)
8. Haase, G., Liebmann, M., Douglas, C., Plank, G.: A parallel algebraic multigrid solver on graphical processing unit. In: W. Zhang, et al. (eds.) HPCA 2010, vol. LNCS 5938, pp. 38–47. Springer, Heidelberg (2010)
9. Hackbusch, W.: Multi-grid methods and applications, 2. print. edn. Springer series in computational mathematics ; 4. Springer, Berlin (2003)
10. Henson, V.E., Yang, U.M.: BoomerAMG: a parallel algebraic multigrid solver and preconditioner. Appl. Numer. Math. **41**(1), 155–177 (2002). DOI http://dx.doi.org/10.1016/S0168-9274(01)00115-5

11. Heuveline, V., et al.: HiFlow^3 - parallel finite element software (2011). URL http://www.hiflow3.org/
12. Heuveline, V., Lukarski, D., Subramanian, C., Weiss, J.P.: Parallel preconditioning and modular finite element solvers on hybrid CPU-GPU systems. In: P. Iványi, B.H. Topping (eds.) Proceedings of the 2nd Int. Conf. on Parallel, Distributed, Grid and Cloud Computing for Engineering, Paper 36. Civil-Comp Press, Stirlingshire, UK (2011). DOI doi:10.4203/ccp.95.36
13. 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, LNCS vol. 6586, pp. 389–397 (2010)
14. Heuveline, V., Lukarski, D., Weiss, J.P.: Enhanced parallel ILU(p)-based preconditioners for multi-core CPUs and GPUs – the power(q)-pattern method. Tech. Rep. 2011-08, EMCL, KIT (2011). URL http://www.emcl.kit.edu/preprints/emcl-preprint-2011-08.pdf
15. 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)
16. Hülsemann, F., Kowarschik, M., Mohr, M., Rüde, U.: Parallel geometric multigrid. In: Numerical Solution of Partial Differential Equations on Parallel Computers, volume 51 of LNCSE, chapter 5, pp. 165–208 (2005)
17. Kolotilina, L., Yeremin, A.: Factorized sparse approximate inverse preconditionings, I: theory. SIAM J. Matrix Anal. Appl. (1993)
18. Saad, Y.: Iterative methods for sparse linear systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003)
19. Shapira, Y.: Matrix-based multigrid. Theory and applications (Numerical methods and algorithms). Springer (2003)
20. Trottenberg, U.: Multigrid. Academic Press, Amsterdam (2001). Academic Press
21. Wittum, G.: On the robustness of ILU smoothing. SIAM Journal on Scientific and Statistical Computing, 10:699717 (1989)