Block-asynchronous and Jacobi smoothers for a multigrid solver on GPU-accelerated HPC clusters

Martin Wlotzka, Vincent Heuveline

Preprint No. 2015-03

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
Preprint No. 2015-03

Affiliation of the Authors

Martin Wlotzka[a,1], Vincent Heuveline[a]

[a] *Engineering Mathematics and Computing Lab (EMCL), Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, Germany*
[1] *Corresponding Author: Martin Wlotzka, martin.wlotzka@uni-heidelberg.de*

Impressum

Heidelberg University
Interdisciplinary Center for Scientific Computing (IWR)
Engineering Mathematics and Computing Lab (EMCL)

Speyerer Str. 6,
69115 Heidelberg
Germany

Block-asynchronous and Jacobi smoothers for a multigrid solver on GPU-accelerated HPC clusters

Martin Wlotzka, Vincent Heuveline

Abstract

We investigate CPU- and GPU-based damped block-asynchronous iteration as an alternative for the damped CPU-based Jacobi smoother in a geometric multigrid linear solver. We depict the implementation for distributed memory systems as well as for CUDA-capable accelerators. Our numerical experiments are based on the linear problem arising from a finite element discretization of the Poisson equation. Runtime and energy measurements are presented for a dual-CPU test system equipped with a GPU. We find that the smoothing properties of the block-asynchronous smoothers are diminished by their asynchronous nature. When using a domain decomposition, damped synchronized Jacobi iteration as smoother with CPU-only computation on multiple host processes yields better performance and lower energy consumption than the block-asynchronous variants for both CPU and GPU execution. However, for a single host process without domain decomposition, the GPU-accelerated block-asynchronous method can compensate the diminished smoothing property and outperforms the CPU-only execution both in terms of runtime and energy consumption.

1 Introduction

Multigrid solvers belong to the most efficient numerical methods for solving symmetric positive definite linear systems. The computational complexity is $O(n)$ for sparse systems with n unknowns. The efficiency of a multigrid method strongly depends on the smoothing method employed. The role of the smoother is to remove high frequency error contributions from the solution. Classical relaxation schemes such as Jacobi or Gauss-Seidel iteration and their damped variants are often used as smoothers. In the context of high-performance computing (HPC), scalability of all building blocks of the multigrid solver is crucial for good parallel performance. At the same time, the smoothing properties must be maintained to sustain the efficiency.
In this work, we briefly present our geometric multigrid solver which is parallelized for distributed memory HPC clusters. The multigrid framework is seamlessly integrated in the general purpose parallel finite element package *HiFlow3* [4]. The main part of this work is dedicated to an investigation of the usefulness of a damped block-asynchronous multigrid smoother as opposed to a classical damped Jacobi smoother on a GPU-accelerated distributed memory host system. Asynchronous methods have the potential of exploiting parallel hardware architectures such as multi-core CPUs, many-core accalerators or independent host processes more efficiently thanks to their relaxed synchronization requirements. By conducting a series of numerical experiments, we address the question whether these methods still show suitable smoothing properties in spite of their asynchronous nature. The tests are based on the finite element solution of the 2D Poisson problem, a prototypical second order elliptic partial differential equation.

1.1 Related work and paper contribution

The idea of "chaotic relaxation" was proposed by Rosenfeld [17], who used "parallel-processor computing systems" to simulate the distribution of current in an electrical network. Chazan and Miranker in 1969 [7]

were the first to study this type of methods on a rigorous theoretical basis. They established a charaterization of the chaotic relaxation schemes for the solution of symmetric positive definite linear problems and gave conditions for convergence, as well as examples for divergence. Meanwhile, the denomination "asynchronous iteration" has been established in the literature for this type of methods. An overview of asynchronous schemes and convergence theory can be found in [10]. Asynchronous iteration has successfully been used in the context of high-performance computing, see e.g. [8] and references therein.
For GPU-accelerated systems, works reported in [2] and [3] investigate numerical convergence and performance of block-asynchronous iteration, both as plain solver and in combination with mixed precision iterative refinement. The usage of block-asynchronous smoothers in the context of multigrid methods has been investigated in [1]. However, these works are restricted to single node configurations and the host CPUs are not taken into account for block-asynchronous computations. We extend this setup to the case of distributed memory machines with several host processes sharing a device. Additionally, we compare to asynchronous CPU-only smoothers which also benefit with respect to parallel performance from relaxed synchronization requirements. Finally, we assess runtime performance and conduct actual energy measurements to investigate the energy consumption of the methods.

1.2 Paper organization

This paper is structured in the following way: We outline the experimental setup describing the test problem and the numerical methods in Section 2. In particular, we present the main features of our geometric multigrid implementation and the smoothers using MPI [13] for distributed memory systems and CUDA [15] for graphics processing units (GPUs). Section 3 is devoted to the discussion of the results, and Section 4 concludes the work.

2 Experimental setup

In this work, we investigate the performance and energy consumption of a parallel geometric multigrid linear solver using the damped Jacobi method as well as damped asynchronous iteration schemes as smoothers. Our test problem is the 2D Poisson equation discretized by Lagrange finite elements. In this section, we describe the mathematical background of the methods and the test problem.

2.1 Linear problem

In our experiments, we use the linear system of equations arising from a finite element discretization of the two-dimensional Poisson equation. This equation can be used to model the equilibrium heat distribution in a physical domain $\Omega \subset \mathbb{R}^2$ with given environmental temperature and heat sources or sinks. The problem definition reads

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \ , \\ u &= g \quad \text{on } \partial\Omega_{\mathrm{D}} \ , \\ \nabla u \cdot n &= 0 \quad \text{on } \partial\Omega_{\mathrm{N}} \ , \end{aligned}$$

where f represents any heat sources or sinks inside the domain and g is the environmental temperature given through the Dirichlet condition on the boundary part $\partial\Omega_{\mathrm{D}}$. Thermal insulation is modeled by the homogeneous Neumann boundary condition on the boundary part $\partial\Omega_{\mathrm{N}}$. We use Lagrange finite elements to discretize the partial differential equation on a mesh Ω_h leading to the linear system $A_h x_h = b_h$ with

$$\begin{aligned} (A_h)_{ij} &= \int_\Omega \nabla(\phi_h)_j \cdot \nabla(\phi_h)_i \, dx \\ b_i &= \int_\Omega f(\phi_h)_i \, dx \ , \quad u = \sum_{i=1}^{n} x_i(\phi_h)_i \ , \end{aligned}$$

where the $(\phi_h)_i$ $(i = 1, ..., n)$ denote the finite element basis functions on Ω_h. The resulting system matrix A_h is symmetric and positive definite [9]. It is also called stiffness matrix and b_h the load vector

Algorithm 1 Basic linear iteration scheme

```
1: Set initial solution $x^0$, tolerance $\epsilon > 0$, $n = 0$.
2: Compute the residual $r^0 = b - Ax^0$.
3: while $\|r^n\| > \epsilon$ do
4:     Solve $Ae = r^n$ approximately: $\hat{e} = Br^n$
       with $B \approx A^{-1}$.
5:     Update $x^{n+1} = x^n + \hat{e}$.
6:     Compute the residual $r^{n+1} = b - Ax^{n+1}$.
7:     $n \leftarrow n + 1$
8: end while
```

for historical reasons, when this type of discretization was applied to elasticity problems. The matrix A_h is sparse due to the small support of the Lagrange finite element basis functions embracing only mesh cells which are adjacent to the corresponding Lagrange point.
For our experiments, we choose the unit square $\Omega = (0,1) \times (0,1)$, $\partial\Omega_{\mathrm{D}}$ to be the left and right side and $\partial\Omega_{\mathrm{N}}$ to be the top and bottom side of Ω. The source term is defined as $f(x,y) = 200x(1-x)y(1-y)$, and the boundary value is

$$g(x,y) = \begin{cases} 1 + 3y(1.5 - y) & \text{if } x = 0 \ , \\ (y + 0.5)(y - 1) & \text{if } x = 1 \ . \end{cases}$$

Figure 1 shows a visualisation of the solution to this Poisson problem.

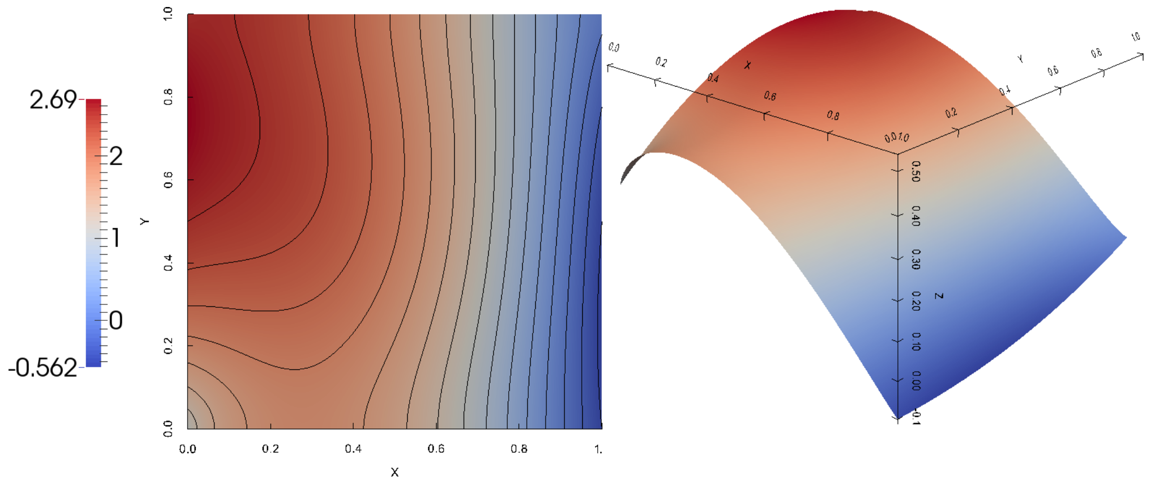

Figure 1: Visualization of a finite element solution for the Poisson equation.

2.2 Geometric multigrid linear solver

Our geometric multigrid linear solver belongs to a group of algorithms that can be described in a very general framework of linear iterative schemes [20]. In an abstract setup, the goal is to solve a linear system

$$Ax = b \ ,$$

where A is a symmetric positive definite operator on a finite-dimensional vector space V. A linear iterative scheme which uses an old approximation x^n to compute a new approximation x^{n+1} can often be characterized by the steps in Algorithm 1. This scheme is also called iterative refinement method. The algorithm grants full flexibility with respect to the solution of the error equation in step 4. The idea of multilevel methods is to compute the error correction in a space $\hat{V}$ of smaller dimension $\dim(\hat{V}) < \dim(V)$

[11]. In the context of finite element discretizations of partial differential equations, the spaces V and $\hat{V}$ may result from discretizations on a fine and a coarse grid, respectively. In this case, the scheme is also called geometric multigrid method to emphasize its construction from the discretization of the problem on different grids. A simple way to contruct the spaces is by uniform refinement of a coarse grid Ω_{2h} yielding the fine grid Ω_h and corresponding spaces $\hat{V} = V_{2h}$ and $V = V_h$. The transition between the two grid levels h and $2h$ is defined by a prolongation and a restriction operator. The prolongation operator $P_{2h}^h : V_{2h} \to V_h$ maps a vector from the coarse grid to the fine grid. In our method, we use linear interpolation for the prolongation. The restriction operator $R_{2h}^h : V_h \to V_{2h}$ maps a vector from the fine grid to the coarse grid. We choose the restriction to be the transpose of the interpolation. A crucial ingredient of multigrid methods is the smoother. Its purpose is to remove high frequency error contributions on the fine grid, so that the smoothed error can be represented on the coarse grid. Relaxation schemes such as Jacobi or Gauss-Seidel iteration and their damped variants are often used as smoothers. For efficient smoothing methods, often a small number $\mu \leq 3$ of smoother iterations is sufficient to damp out the high frequencies. In this work, we investigate the applicability of asynchronous iteration schemes as smoothers and compare them to classical synchronized Jacobi-type iteration.
Note that Algorithm 1 can be applied recursively. For our geometric multigrid method, this amounts to choosing a number L of levels and a coarsest grid parameter $H > 0$, yielding a grid hierarchy $\{\Omega_h \,|\, h = H/2^{l-1}, l = 1, 2, ..., L\}$ and corresponding finite element spaces V_h. The operator A and the vectors x and b from the abstract scheme have to be replaced by their analogons A_h, x_h and b_h on the corresponding grid level. One solution update cycle of the geometric multigrid method is stated in Algorithm 2. It is characterized by the number γ of recursive cycle calls. The choices $\gamma = 1$ and $\gamma = 2$ lead to the V- and W-cycle, respectively, depicted in Fig. 2. The term $S_h(A_h, x_h, b_h, \mu)$ indicates the execution of μ smoothing iterations on the coresponding grid level. On the coarsest grid, the error correction equation is solved with high accuracy. In our setup, we use the Conjugate Gradient [18] method as coarse grid solver.

Algorithm 2 Cycle($A_h, x_h, b_h, \gamma, \mu$)

```
1:  if $h = H$ then
2:      $x_h \longleftarrow A_h^{-1} b_h$ (coarse grid solution)
3:  else
4:      $x_h \longleftarrow S_h(A_h, x_h, b_h, \mu)$ (pre-smoothing)
5:      $r_h \longleftarrow b_h - A_h x_h$ (residual computation)
6:      $b_{2h} \longleftarrow R_{2h}^h(r_h)$ (restriction)
7:      $x_{2h} \longleftarrow 0$
8:      for $k = 1, 2, ..., \gamma$ do
9:          Cycle($A_{2h}, x_{2h}, b_{2h}, \gamma, \mu$) (recursion)
10:     end for
11:     $c_h \longleftarrow P_{2h}^h(x_{2h})$ (prolongation)
12:     $x_h \longleftarrow x_h + c_h$ (correction)
13:     $x_h \longleftarrow S_h(A_h, x_h, b_h, \mu)$ (post-smoothing)
14: end if
```

The number of smoother iterations executed on a certain level l withing one cycle is given as

$$\mu(\gamma, l) = 2\mu\gamma^{l-1} \quad (l < L) \tag{1}$$

when counting from the finest level $l = 1$ to the coarsest level $l = L$. Note that the smoother is not active on the coarsest grid itself.

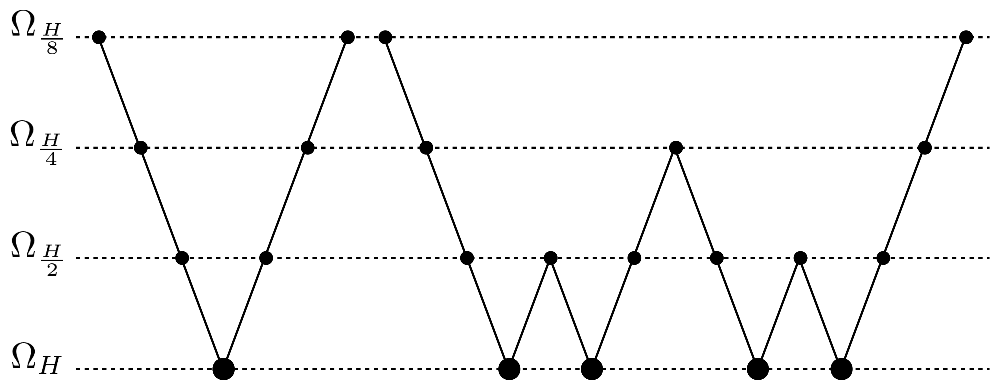

Figure 2: Visulaization of V- and W-cycle on four grids. Small dots indicate smoothing, larger dots indicate coarse grid solving.

2.3 Damped Jacobi and block-asynchronous iteration smoothers

The goal of this work is to investigate the applicability of parallel asynchronous iteration schemes for smoothing as an alternative to synchronized methods like the Jacobi iteration. The asynchronous iteration methods considered in this work can be derived from the classical Jacobi relaxation method [3]. In the context of multigrid methods, often the damped variants of the relaxation schemes are used, since they show better smoothing properties in many cases [21]. Assuming the diagonal part D of the system matrix $A \in \mathbb{R}^{n\times n}$ to be regular, the damped Jacobi iteration with damping parameter $\omega > 0$ reads [12]

$$\begin{aligned} x^{k+1} &= x^k + \omega D^{-1}\Big[b - Ax^k\Big] \\ &= B(\omega)x^k + d(\omega) \end{aligned} \qquad (k = 0, 1, ...)\ , \tag{2}$$

where $B(\omega) = I - \omega D^{-1}A$ is the iteration matrix and $d(\omega) = \omega D^{-1}b$.
The parallelization of this method is straightforward. Each compute unit, may it be a process in a distributed system, a thread on an accelerator device, or a hybrid form of the aforementioned, may compute a part of the new iteration vector x^{k+1}. More precisely, let $I \subset \{1, ..., n\}$ be the index set of all components of the iteration vector which are computed by one such compute unit. The componentwise form of the damped Jacobi iteration for that part of the vector reads

$$i \in I \ : \quad x_i^{k+1} = x^k + \frac{\omega}{a_{ii}}\Big[b_i - \sum_{j=1}^{n} a_{ij}x_j^k\Big]\ .$$

Note that for computing its part of the new iterate x^{k+1}, any compute unit potentially uses components of the preceding iterate x^k which belong to other compute units. This requires a synchronization of the compute units after each iteration to make sure that all needed values are updated from the last iteration. The idea of asynchronous iteration is to overcome the synchronization requirements. On the theoretical level, this is accomplished by introducing a shift function s and an update function u in the iteration:

$$x_i^{k+1} = \begin{cases} x_i^k + \frac{\omega}{a_{ii}}\Big[b_i - \sum_{j=1}^{n} a_{ij}x_j^{k-s(j)}\Big] & \text{if } i = u(k) \\ x_i^k & \text{if } i \neq u(k) \end{cases}$$

The shift function allows to use not only values from the last iteration, but also older or newer values. The update function chooses one component at a time to be updated, leaving the other components unchanged [1]. A sufficient condition for convergence is uniform boundedness of s, and u must take each value in $\{1, ..., n\}$ infinitely often, and $\rho(|B(\omega)|) < 1$ [7].
A natural modification of the basic asynchronous scheme is the aggregation of components into blocks [5].

Let P be the number of blocks, and $I_p \subset \{1, ..., n\}$ be the index set of all components belonging to block $p \in \{1, ..., P\}$. The block-asynchronous iteration reads

$$i \in I_p \; : \quad x_i^{k+1} = x_i^k + \frac{\omega}{a_{ii}} \Big[b_i - \sum_{j \notin I_p} a_{ij} x_j^{k-s(k,j)} - \sum_{j \in I_p} a_{ij} x_j^k \Big] \, .$$

This scheme is synchronized only with respect to the vector components within each block. The block scheme implies a decomposition of the systems matrix A into diagonal and offdiagonal parts

$$D_p = \big(a_{ii}\big)_{i \in I_p}, A_p^{\text{diag}} = \big(a_{ij}\big)_{i,j \in I_p}, A_p^{\text{offdiag}} = \big(a_{ij}\big)_{i \in I_p, j \notin I_p}$$

and a decomposition of the vectors x and b into local parts

$$\begin{aligned} x_p^{\text{local}} &= \big(x_i\big)_{i \in I_p}, \; b_p^{\text{local}} = \big(b_i\big)_{i \in I_p}, \\ x_p^{\text{non-local}} &= \big(x_j\big)_{j \notin I_p, \exists i \in I_p \, : \, a_{ij} \neq 0} \, . \end{aligned}$$

Such block decomposition is sketched for A and x in Figure 3. The update step for any block l then reads

$$x_p^{\text{local}} \leftarrow x_p^{\text{local}} + \omega D_l^{-1} \Big[b_p^{\text{local}} - A_p^{\text{diag}} x_p^{\text{local}} - A_l^{\text{offdiag}} x_p^{\text{non-local}} \Big] \, .$$

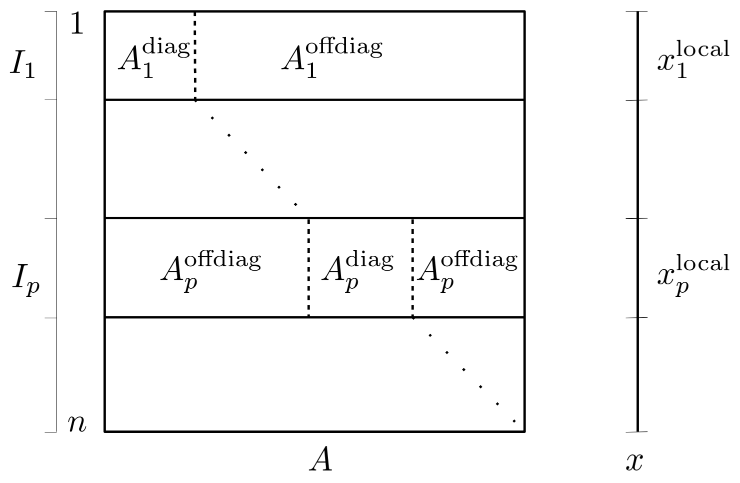

Figure 3: Decomposition of the system matrix A and solution vector x into blocks.

The block-asynchronous scheme can be extended by performing multiple iterations on the local block before updating the values in the non-local vector part. We denote the resulting algorithm as *damped async-(m^{outer}, m^{inner})* to indicate m^{outer} outer iterations with non-local vector updates and m^{inner} inner asynchronous steps for each outer iteration. Obviously, each block can be mapped to one compute unit, resulting in Algorithm 3. In terms of the smoother denomination $S_h(A_h, x_h, b_h, \mu)$ in Algorithm 2, the number of smoothing iterations for the block-asynchronous method is $\mu = m^{\text{outer}} \times m^{\text{inner}}$. Accordingly, we indicate by *damped Jacobi-(μ)* the use of μ synchronous damped Jacobi iterations as defined in Eq. (2).

2.4 Implementation

Our implementation spans two levels of parallelism. It supports multi-node distributed memory systems where the nodes are connected by a network. Communication between the nodes is done by data transfer over the network using MPI [13]. On the node level, it supports CUDA-capable devices [15].
The implementation is integrated in the HiFlow^3 package [4]. It uses a decomposition of the computational domain into subdomains and corresponding MPI-parallelized matrix and vector data structures. The

Algorithm 3 damped async-(m^{outer}, m^{inner})

```
1: for $k^{\text{outer}} = 1, 2, ..., m^{\text{outer}}$ do
2:     Update non-local vector parts with corresponding values from other blocks.
3:     for all blocks $p = 1, ..., P$ in parallel do
4:         for $k^{\text{inner}} = 1, 2, ..., m^{\text{inner}}$ do
5:             $x_p^{\text{local}} \leftarrow x_p^{\text{local}} + \omega D_p^{-1} \Big[ b_p^{\text{local}} - A_p^{\text{diag}} x_p^{\text{local}} - A_p^{\text{offdiag}} x_p^{\text{non-local}} \Big]$
6:         end for
7:     end for
8: end for
```

matrix and the vectors are distributed among the MPI processes, thus defining the block decomposition. The communication pattern is derived from the matrix structure and avoids any unnecessary data transfer. Only vector components corresponding to non-zero entries in the offdiagonal matrix blocks of other MPI processes are transferred.
We use matrix-based prolongation and restriction operators which exploit the MPI-parallelized data structures, as well as the coarse grid CG solver. For the smoother, the parallelism of the local block updates in Algorithm 3 corresponding to steps 3-7 is achieved by concurrency of the MPI processes. All computations of the smoother are either executed on the host CPUs, or on the accelerator devices. The update step 2 implies MPI communication and, if the CUDA version is used, data transfer between host and device. One or multiple MPI processes may be scheduled onto each node. Within each node, the MPI

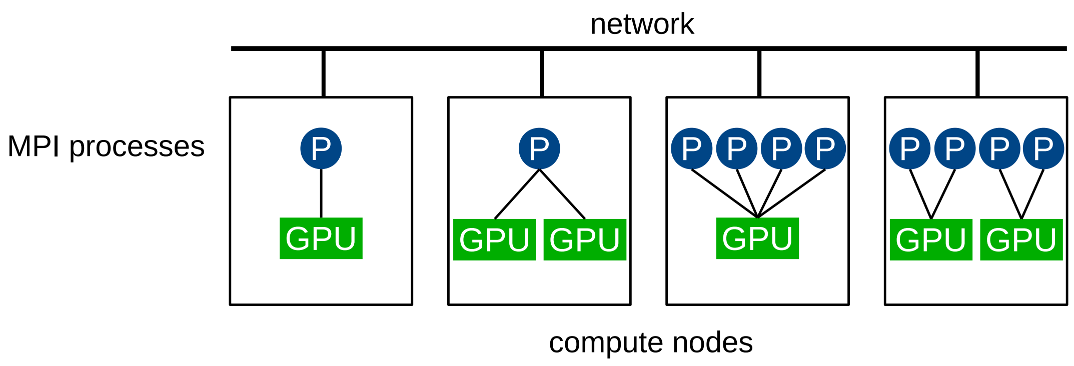

Figure 4: Supported configurations of MPI process scheduling among compute nodes and GPU usage.

processes can use multiple GPUs. The actual utilization may be configured depending on the number of MPI processes and on the number of available devices. If only one MPI process is scheduled onto a node, this process may use all available devices on that node, as sketched in the two left configurations in Figure 4. In case of multi-GPU usage of a single MPI process, the matrix and vector blocks of this process are further split into sub-blocks as depicted in Figure 5. However, if multiple MPI processes are scheduled onto the same node, GPU utilization must be split such that each process uses only one of the available devices, see the two right configurations in Figure 4. This limitation is imposed by the Nvidia multi-process service (MPS) [16], which is necessary to maintain the HyperQ feature of Kepler GPUs [14]. A practical way for preparing the host system to meet this technical requirement is described in [19].
Our geometric multigrid solver is implemented in C++ for execution on CPUs, including the damped

Jacobi and block-asynchronous iteration smoothers. In addition, there is an alternative implementation of the block-asynchronous iteration smoother in CUDA for execution on accelerators.

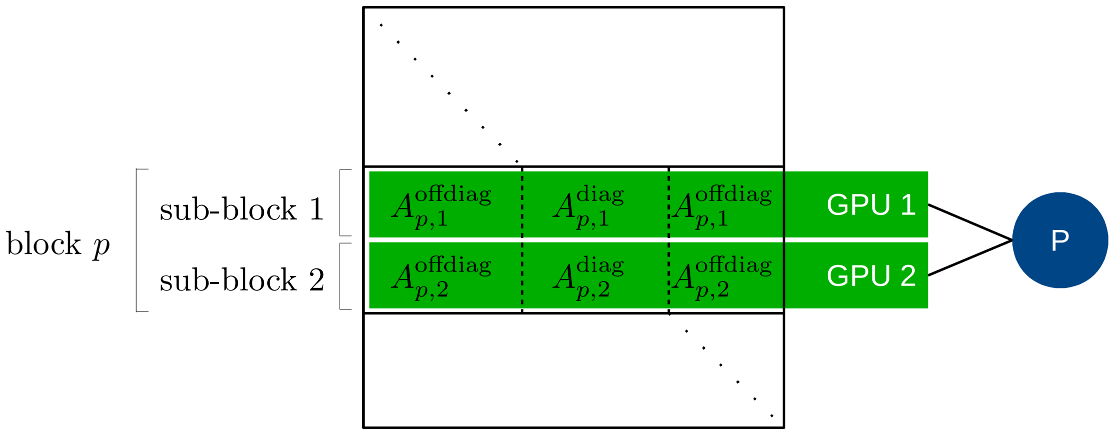
block p
sub-block 1
sub-block 2
$A_{p,1}^{\text{offdiag}}$
$A_{p,1}^{\text{diag}}$
$A_{p,1}^{\text{offdiag}}$
$A_{p,2}^{\text{offdiag}}$
$A_{p,2}^{\text{diag}}$
$A_{p,2}^{\text{offdiag}}$
GPU 1
GPU 2
P

Figure 5: Sub-block decomposition in case of multi-GPU usage by a single MPI process.

2.5 Parameter definitions

The parameter definitions for our numerical experiments are detailed in Table 1.

Table 1: Parameter definitions.

finite elements	Lagrange Q2
finest grid $l = 1$	1,050,625 unknowns
$l = 2$	263,169 unknowns
$l = 3$	66,049 unknowns
coarsest grid $l = 4$	16,641 unknowns
fine grid residual tolerance	$\epsilon = 10^{-12}$
coarse grid CG tolerance	$\delta = 10^{-13}$
cycle type	W-cycle ($\gamma = 2$)
damping	$w = 0.5$

2.6 Hardware, build system, and measurement system

Our test system consisted of one compute node equipped with 2 x Intel Xeon E-4650, 512 GByte DDR3 main memory and an Nvidia Tesla K40. We used GCC compiler version 4.8.2, OpenMPI version 1.6.5, CUDA version 6.5.12, and NVIDIA device driver version 340.65.
For power measurement, we used the ZES Zimmer Electronic Systems LMG450 external power meter. The sensors were attached to the input lines of the power supply units of the node. Thus, we measured the total power consumption of the whole node. We used the maximum possible sampling rate of 20 Hz of the LMG450 power meter. The measurement was controlled using the `pmlib` tool [6]. We instrumented the solver code using the `pmlib` client API to measure exactly that portion of the overall program which constitutes the solution process. This excluded all initialization overhad from the measurements. The `pmlib` server ran on a separate machine to avoid a perturbation of the system under investigation. The setup is shown in Figure 6.

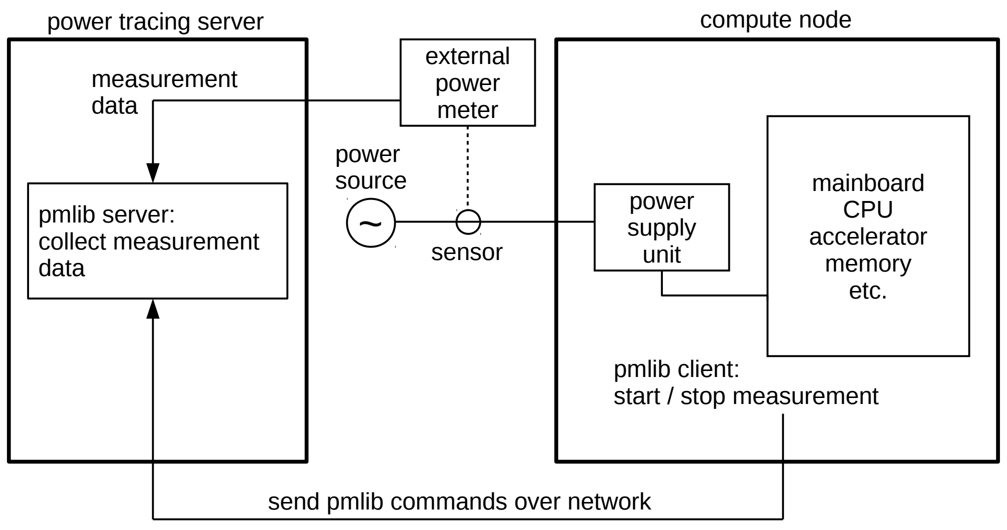
power tracing server
compute node
measurement data
external power meter
power source
sensor
pmlib server: collect measurement data
power supply unit
mainboard CPU accelerator memory etc.
pmlib client: start / stop measurement
send pmlib commands over network

Figure 6: Measurement setup using an external power meter controlled by the pmlib tool.

3 Results

We carried out three series of tests for our geometric multigrid solver with different smoother methods:

1. *CPU damped Jacobi-(3)* Geometric multigrid solver using a damped synchronous Jacobi smoother which runs on the host CPUs, 3 smoothing steps.
2. *CPU damped async-(3,5)* Geometric multigrid solver using a damped block-asynchronous iteration smoother which runs on the host CPUs, 3 outer smoothing steps with non-local vector updates, and 5 asynchronous inner iterations for each outer step.
3. *GPU damped async-(3,5)* Geometric multigrid solver using a damped block-asynchronous iteration smoother which runs on the GPU, 3 outer smoothing steps with non-local vector updates, and 5 asynchronous inner iterations for each outer step.

We defined these test series to evaluate the effect on performance and energy consumption of using an asynchronous iteration scheme in contrast to the synchronous damped Jacobi smoother. In addition, our goal is to investigate the effect of executing the asynchronous smoother on accelerators instead of CPU-only computations. For each of the three test series, we executed the computation with varying number of MPI processes $p = 1, 2, 4, 8, 16$. We repeated all experiments five times and report average results in Table 2. Time and energy are given in seconds and Watt-seconds, respectively, rounded to three decimals. In the last three columns of the table, we report the number of W-cycles needed to reach the desired accuracy, the accumulated number of coarse grid CG iterations and the accumulated number of smoothing iterations. According to Eq. (1), the accumulated number of smoother iterations m is determined as

$$m = (\text{no. W-cycles}) \times \sum_{l=1}^{3} \mu(2, l) \; .$$

For the asynchronous smoother methods, the accumulated number of coarse grid CG iterations represents the average from the five repetitions, rounded to the next integer. In addition, we report time-to-solution and number of CG iterations for a plain CG solver on the finest grid in the lower block of the table.

Table 2: Measurement results.

method	p	time [s]	energy [Ws]	cyc.	CG iter.	smoother
CPU damped	1	6.597	3,051.213	11	2,523	462
Jacobi-(3)	2	3.543	1,779.145	11	2,523	462
	4	1.868	1,040.641	11	2,523	462
	8	1.035	629.727	11	2,523	462
	16	0.706	470.709	11	2,523	462
CPU damped	1	6.898	3,106.624	4	1,421	840
async-(3,5)	2	6.386	3,152.513	7	2,877	1,470
	4	3.093	1,792.762	7	3,014	1,470
	8	1.745	1,132.060	7	3,078	1,470
	16	1.223	863.091	7	3,187	1,470
GPU damped	1	4.845	2,465.331	7	2,872	1,470
async-(3,5)	2	4.121	2,264.939	7	3,165	1,470
	4	3.325	1,961.846	7	3,229	1,470
	8	2.840	1,781.422	7	3,264	1,470
	16	2.669	1,771.580	7	3,319	1,470
plain CG	1	46.113			1,620	
on CPU	2	23.712			1,620	
	4	13.003			1,620	
	8	7.074			1,620	
	16	5.283			1,620	

The test series *CPU damped Jacobi-(3)* yielded a number of 11 W-cycles with 462 accumulated smoother iterations and 2,523 accumulated coarse grid CG iterations. As expected, these numbers were constant among all parallel configurations, since the smoother method used in this test series is synchronous. The runs showed a reduction in runtime from 6.597 seconds to 0.706 seconds for p ranging from 1 to 16, see also Fig. 7. However, speedups were clearly inferior than theoretical linear speedup. The reason lies in the smaller problem sizes of the coarser grids in the hierarchy. Since the majority of the overall number of smoother iterations is executed on coarser grids, and since all CG iterations are performed on the coarsest grid, the poorer parallel efficiency for smaller problem sizes affects the overall performance. Nevertheless, comparing with the runtimes from the lower block on the table, the performance of our geometric multigrid solver is by far superior over the plain fine grid CG solver. The test series *CPU damped async-(3,5)* contains the special case $p = 1$. In this case, the smoother method is

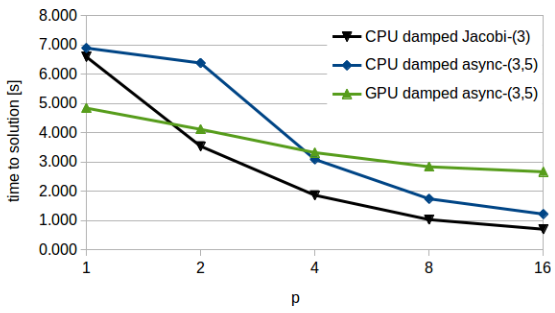

Figure 7: Time to solution in seconds.

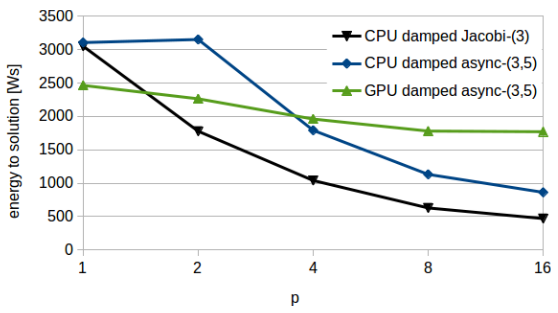

Figure 8: Energy to solution in Watt-seconds.

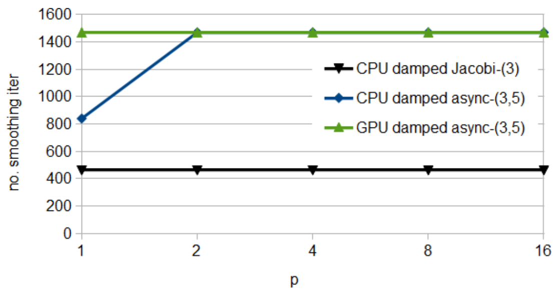

Figure 9: Accumulated number of smoothing iterations.

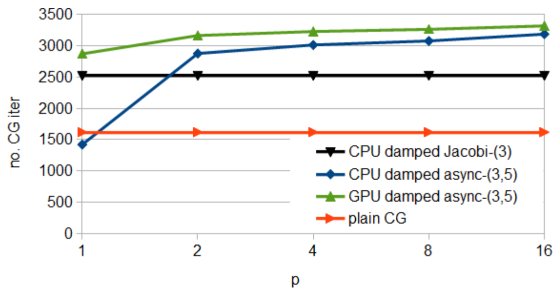

Figure 10: Accumulated number of conjugate gradient iterations.

actually equivalent to a synchronous damped Jacobi-(15) smoother, since the whole problem resides on a single process and no asynchronism is present. This resulted in a much smaller number of 4 W-cycles since the smoother is doing five times the number of iterations per cycle compared to *CPU damped Jacobi-(3)*. In total, the accumulated number of smoother iterations was 840, and the accumulated number of coarse grid CG iterations was 1,421. Although this meant a decrease of CG iterations by 1,102 while the increase of smoother iterations was only 378, the time-to-solution of 6.898 seconds was similar to the run with $p = 1$ from *CPU damped Jacobi-(3)*. This is due to the fact that the smoother is executed on the grid levels $l = 1, 2, 3$ where the problem size is a factor 4^{4-l} larger than on the coarsest grid, where the CG iterations were saved.
In the other cases of test series *CPU damped async-(3,5)* with $p = 2, 4, 8, 16$, the introduction of asynchronism diminished the smoothing ability of the smoother method. This is reflected by an increased number of 7 W-cycles with 1,470 accumulated smoother iterations and an increased number of accumulated coarse grid CG iterations between 2,877 and 3,187. Accordingly, the runtimes were larger than in the test series *CPU damped Jacobi-(3)*.
All runs from the *GPU damped async-(3,5)* test series yielded 7 W-cycles with 1,470 accumulated smoother iterations, which is the same as for *CPU damped async-(3,5)* with $p \geq 2$. This is clearly due to the asynchronism in the smoother method. Also for $p = 1$, the execution of the smoother on the device has a block-asynchronous nature, since the threads on the GPU are only synchronized with respect to their CUDA thread block. Analogously, the accumulated number of coarse grid CG iterations was on a similar level as for the asynchronous CPU-only test runs, namely ranging from 2,872 for $p = 1$ to 3,319 for $p = 16$.
The *GPU damped async-(3,5)* run with $p = 1$ showed the best performance among all three test series for the single MPI process configuration. The execution of the block-asynchronous smoother on the GPU with 4.845 seconds time-to-solution rendered a speedup factor of ≈ 1.36 over CPU-only computation. However, the GPU-accelerated runs showed only small speedup when using more MPI processes. This is due to the fact that the GPU always carries out the total smoother computations of all MPI processes, such that the total problem size on the GPU is constant for all p. Also, GPU-usage imposes an overhead for data transfer between host and device. Only the grid transfer operators and the coarse grid CG solver can benefit from a greater number of MPI processes. As a consequence, in all the cases $p = 2, 4, 8, 16$ the *CPU damped Jacobi-(3)* runs showed the best performance.
The energy consumption, plotted in Fig. 10, was strongly correlated to the runtimes. Also here, the usage of the GPU-accelerated asynchronous smoother method yielded the lowest energy consumption for the $p = 1$ case among all three test series, while for $p \geq 2$ usage of the synchronous CPU-based smoother consumed least energy.

4 Conclusion

We investigated a geometric multigrid linear solver using synchronous and asynchronous variants of a damped Jacobi smoother. We introduced the mathematical background of the multigrid solver and of the smoother methods, which were derived from the classical Jacobi method. We presented our implementation of these methods with support for distributed memory systems by means of an MPI parallelization, as well as support of CUDA-capable devices. We ran three series of tests on a compute node equipped with

four Intel Xeon E-4650 CPUs and an Nvidia Tesla K40 GPU. We designed the tests to assess the effect of introducing asynchronism in the smoother method for parallel execution in contrast to synchronized methods, and to assess the effect of using accelerators instead of CPU-only computation. We measured the performance in terms of runtime, and we used an external high-precision power meter to measure the energy consumption.
We found that asynchronism generally diminishes the smoothing ability of the smoother method. All test runs with block-asynchronous smoother needed substantially more multigrid cycles and thus smoother iterations than their synchronous pendant to reach the same final accuracy. Using smaller local blocks, and therefore increasing asynchronism through the independent local block updates, further reduces the smoothing effect and results in an increased number of coarse grid CG solver iterations. In all but one cases, this drawback could not be compensated by the reduced synchronization requirements in parallel execution. Only the single host process configuration could benefit from an asynchronous smoother when executing it on the GPU.

Acknowledgement

This work was supported by the European Commission as part of the collaborative project “Exa2Green - Energy-Aware Sustainable Computing on Future Technology”under research grant no 318793.

References

[1] H. Anzt, J. Dongarra, M. Gates, and S. Tomov. Block-asynchronous multigrid smoothers for GPU-accelerated systems. *EMCL Preprint Series*, 15, 2011.

[2] H. Anzt, J. Dongarra, V. Heuveline, and P. Luszczek. GPU-Accelerated Asynchronous Error Correction for Mixed Precision Iterative Refinement. *EMCL Preprint Series*, 17, 2011.

[3] H. Anzt, S. Tomov, J.Dongarra, and V. Heuveline. A block-asynchronous relaxation method for graphics processing units. *Journal of Parallel and Distributed Computing*, 73:1613–1626, 2013.

[4] H. Anzt, F. Wilhelm, J.P. Weiß, C. Subramanian, M. Schmidtobreick, M. Schick, S. Ronnas, S. Ritterbusch, A. Nestler, D. Lukarski, E. Ketelaer, V. Heuveline, A. Helfrich-Schkarbanenko, T. Hahn, T. Gengenbach, M. Baumann, W. Augustin, and M. Wlotzka. HiFlow3: A Hardware-Aware Parallel Finite Element Package. pages 139–151, 2012.

[5] Z.Z. Bai, V. Migallon, J. Penades, and D.B. Szyld. Block and asynchronous two-stage methods for mildly nonlinear systems. *Numerische Mathematik*, 82:1–20, 1999.

[6] S. Barrachina, M. Barreda, S. Catalan, M.F. Dolz, G. Fabregat, R. Mayo, and E.S. Quintana-Orti. An Integrated Framework for Power-Performance Analysis of Parallel Scientific Workloads. In *ENERGY 2013: The Third Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2013.

[7] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

[8] D. ElBaz, A. Frommer, and P. Spiteri. Asynchronous iterations with flexible communication: contracting operators. *J. Comp. Appl. Math.*, 176:91–103, 2005.

[9] A. Ern and J.-L. Guermond. *Theory and Practive of Finite Elements.* Springer, 2004.

[10] A. Frommer and D.B. Szyld. On asynchronous iterations. *J. Comp. Appl. Math.*, 123:201–216, 2000.

[11] W. Hackbusch. *Multi-Grid Methods and Applications.* Springer, 1985.

[12] A. Meister. *Numerik linearer Gleichungssysteme.* Vieweg+Teubner, 2011.

[13] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, Version 3.0, 2012.

[14] NVIDIA Corporation. Kepler GK110, 2012.

[15] NVIDIA Corporation. CUDA Toolkit Documentation v6.5, 2014.

[16] NVIDIA Corporation. Multi-Process Service, 2014.

[17] J. Rosenfeld. A case study in programming for parallel processors. *Communications of the ACM*, 12:645–655, 1969.

[18] Y. Saad. *Iterative Methods for Sparse Linear Systems.* 2000.

[19] F. Wende, T. Steinke, and F. Cordes. Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture. *ZIB Report, Konrad-Zuse-Zentrum für Informationstechnik Berlin*, 2014.

[20] J. Xu. Iterative methods by space decomposition and subspace correction. *SIAM Review*, 34:581–613, 1992.

[21] U.M. Yang. On the use of relaxation parameters in hybrid smoothers. *Numer. Linear Algebra Appl.*, 11:155–172, 2004.

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