ENGINEERING MATHEMATICS
AND COMPUTING LAB

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

An energy-efficient parallel multigrid method for multi-core CPU platforms and HPC clusters

Martin Wlotzka, Vincent Heuveline

Preprint No. 2017-03

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

www.emcl.iwr.uni-heidelberg.de

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
Preprint No. 2017-03

The EMCL Preprint Series contains publications that were accepted for the Preprint Series of the EMCL. Until April 30, 2013, it was published under the roof of the Karlsruhe Institute of Technology (KIT). As from May 01, 2013, it is published under the roof of Heidelberg University.
A list of all EMCL Preprints is available via Open Journal System (OJS) on
http://archiv.ub.uni-heidelberg.de/ojs/index.php/emcl-pp/
For questions, please email to
info.at.emcl-preprint@uni-heidelberg.de
or directly apply to the below-listed corresponding author.

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

Published on the Internet under the following Creative Commons License:
http://creativecommons.org/licenses/by-nc-nd/3.0/de .

An energy-efficient parallel multigrid method for multi-core CPU platforms and HPC clusters

Martin Wlotzka, Vincent Heuveline

June 12, 2017

Abstract

For many applications, multigrid methods can be considered as the method of choice to solve or precondition linear systems of equations in the context of high performance computing. However, maintaining efficiency in the parallelization of the method for distributed memory platforms by means of a domain decomposition is not straight forward due to the different problem sizes of the grids in the multigrid hierarchy. We propose the use of an adaption strategy which adjusts the hardware activity according to the solver needs dynamically during runtime. With a focus on multi-core CPU platforms and clusters, we show that our technique can improve the parallel performance and reduce the energy consumption of the multigrid solver due to a temporary deactivation of CPU cores.

1 Introduction

For many applications, 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. Such systems can result for example from the discretization of elliptic partial differential equations. The geometric multigrid variant [13], as opposed to the algebraic multigrid [12], is based on the discretization of the underlying equations on several grid refinement levels. The operations on fine grid levels, involving smoothers and grid transfer operators, usually employ a limited number of basic numerical building blocks like vector operations or sparse matrix-vector multiplications. Only on the coarsest level, a direct or iterative method is used to solve the error correction equation approximately. For common cluster-level parallelizations based on a domain decomposition which uses the same number of processes throughout the whole grid hierarchy, the overall parallel performance is limited by the parallel performance on the coarsest level with the smallest problem size [7].
The use of multi-core and many-core host systems and co-processors to improve the parallel performance has been studied in many works, see e.g. [4, 6, 7] and references therein. Recently, also the issue of energy consumption has been addressed by means of accelerated smoothers and grid transfer operators [14, 15]. In this work, we address the issues of both parallel performance and energy consumption by using different numbers of processes on different grid levels, thus adapting the parallelization to the problem sizes in the hierarchy. We introduce a dynamic adaption of the hardware activity during the solver execution which adjusts the parallel configuration according to the solver needs.
After a brief presentation of the considered geometric multigrid solver and its parallelization for distributed memory HPC clusters, we propose a concept for the dynamic hardware adjustment during runtime. We assess the effect of the adapted hardware activity both on time and on energy to solution by means of time and power measurements. Our numerical experiments comprise test series with varying total problem size and different hardware adaption strategies on a multi-core compute node equipped with a high precision power meter, and additional performance tests on an HPC cluster using several nodes.

2 Experimental setup

In this section, we describe the mathematical background of the proposed methods assuming a 2D Poisson equation discretized by Lagrange finite elements.

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 [8]. This equation can be used e.g. to model the equilibrium heat distribution in a physical domain 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 $\Omega \in \mathbb{R}$ is the physical domain, f represents any heat sources or sinks 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 $\partial\Omega_{\mathrm{N}}$. For our experiments, we chose the domain Ω to be the unit square. Figure 1 shows a visualization of the solution to this heat problem with non-trivial boundary and source term.

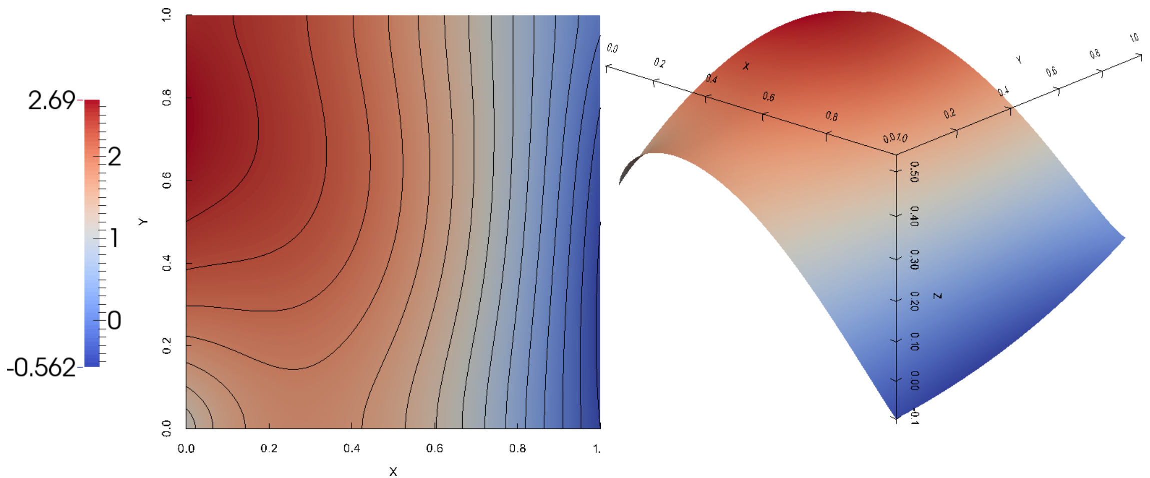

Figure 1: Visualization of the finite element solution for the Poisson equation.

2.2 Geometric multigrid linear solver

Our geometric multigrid linear solver belongs to a class of algorithms that can be described in the very general framework of linear iterative schemes [16]. 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 $\dim(\hat{V}) < \dim(V)$ [9]. 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

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** $\|r^n\| > \epsilon\|r^0\|$ **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**

on different grids. A simple way to construct 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 $V = V_h$, where the refinement parameter h refers to the diameter of the grid cells. The transfer operators between the two grid levels h and $2h$ are 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. The restriction operator $R_{2h}^h : V_h \to V_{2h}$ maps a vector from the fine grid to the coarse grid. In this work, we choose a linear interpolation as the prolongation operator, and the restriction to be its adjoint operator, i.e. $R_{2h}^h = [P_{2h}^h]^\top$.
Besides the grid transfer operators, another crucial ingredient of multigrid methods is the smoother. Its purpose is to remove high frequency error components 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.
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 usual choices $\gamma = 1$ or $\gamma = 2$ lead to the V- or 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 corresponding grid level. On the coarsest grid, the error correction equation is solved with high accuracy. In our setup, we use the Conjugate Gradient [11] method as coarse grid solver. The number of smoother iterations executed on a certain level l within one cycle is given as

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

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.

2.3 Distributed memory parallelization

In this work the parallelization is based on a domain decomposition scheme using MPI [3] on distributed memory machines. For a number p of processes, the computational domain Ω_h of any grid is split into a corresponding number of subdomains. The partitioning is computed with the help of the graph partitioner tool METIS [10] to obtain a balanced decomposition. The domain decomposition implies a block decomposition of matrices and vectors according to the distribution of the degrees of freedom of the finite element space V_h among the processes. Assuming that a process q holds the set I_q of degrees of freedom, matrices are split into diagonal and off-diagonal parts, and vectors are split into local and non-local parts as follows:

$$A_q^{\text{diag}} = \left(a_{ij}\right)_{i,j \in I_q}, A_q^{\text{offdiag}} = \left(a_{ij}\right)_{i \in I_q, j \notin I_q}, \; x_q^{\text{local}} = \left(x_i\right)_{i \in I_q}, \; x_q^{\text{non-local}} = \left(x_j\right)_{j \notin I_q, \exists i \in I_q \,:\, a_{ij} \neq 0}.$$

Note that we omit the h-subscript of matrices and vectors here and in the rest of this subsection for the sake of readability. Each process is responsible for computations on its local vector part, and the non-local

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**

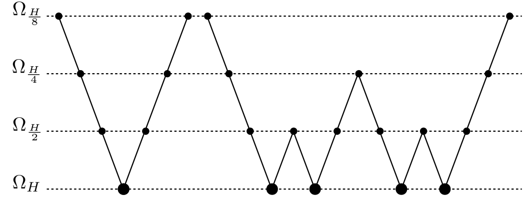

Figure 2: Visualization of V- and W-cycle on four grids. Small dots indicate smoothing, larger dots indicate coarse grid solving.

vector part may contribute to local computations as read-only information. The non-local vector part holds duplicates of components which actually belong to other processes, but which might be needed in local computations due to couplings. The couplings are expressed as non-zero entries in the off-diagonal matrix part. Therefore, whenever non-local vector parts contribute to local computations, they need to be updated with the recent values from the other processes through MPI communication. This parallel setup implies that AXPY-like operations [2] can be performed independently on each process since only the local vector parts are involved:

$$(x + y)_q^{\text{local}} = x_q^{\text{local}} + y_q^{\text{local}} .$$

However, the matrix vector multiplication

$$(Ax)_q^{\text{local}} = A_q^{\text{diag}} x_q^{\text{local}} + A_q^{\text{offdiag}} x_q^{\text{non-local}}$$

requires an update of the non-local vector part and hence communication with other processes. Still, due to the local support of the finite element basis functions, couplings appear only between processes with adjacent subdomains. Therefore, non-local vector updates do mostly not imply a global synchronization, but rather a limited number of peer-to-peer data transfers with the geometric neighbor processes in the domain decomposition. Also, the matrix vector multiplication offers an opportunity to overlap computation and communication according to the following scheme:

1: call non-blocking `MPI_Isend`/`MPI_Irecv` routine for non-local vector update
2: compute $y_q^{\text{local}} \longleftarrow A_q^{\text{diag}} x_q^{\text{local}}$
3: wait for completion of the non-local vector update
4: compute $y_q^{\text{local}} \longleftarrow y_q^{\text{local}} + A_q^{\text{offdiag}} x_q^{\text{non-local}}$

In contrast, the computation of the scalar product of two vectors indeed implies a global synchronization since it represents a global reduction operation:

$$x \cdot y = \sum_{q=1}^{p} x_q^{\text{local}} \cdot y_q^{\text{local}}$$

Looking at the complete multigrid hierarchy, one could in principle use individual domain decompositions with individual numbers of processes for each grid level. However, inefficient communication patterns

could result in the general case. In particular, the efficiency of the prolongation and restriction may suffer if the subdomains on successive grids in the hierarchy are totally unaligned. Additionally, it might not always be beneficial to use all available processes on all grid levels due to the decreasing problem size on coarser grids. It might rather happen that the coarse grid problem size becomes so small that the parallel efficiency drops significantly. We describe our approach to address these issues in the next subsection.

2.4 Dynamic adjustment of hardware activity

We pursue a twofold goal with our parallelization strategy for the full multigrid hierarchy. On the one hand, we intend to use an appropriate number of processes on each grid level, so that particularly the coarse grids do not get fragmented into too small parts in the domain decomposition. On the other hand, we seek to maintain the local communication patterns which exist within each grid level also across the hierarchy for the prolongation and restriction. The desired result is an improved parallel performance and energy consumption of the overall multigrid algorithm.

To meet these goals, we impose restrictions on the general parallelization scheme which was described above. Let L be the number of grid levels in the hierarchy as already denoted above, with $l = 1$ being the finest level and $l = L$ the coarsest. The processes taking part in the domain decomposition on grid l form the MPI communicator $\boldsymbol{c}_l$. Let $\Omega_l^q \subset \Omega_l$ denote the subdomain held by process $q \in \boldsymbol{c}_l$ on grid level l. The restrictions can be stated in the following abstract condition:

$$l = 1, ..., L-1 \;:\quad \forall\, q \in \boldsymbol{c}_{l+1}\, \exists\, \hat{\boldsymbol{c}} \subseteq \boldsymbol{c}_l \quad \text{such that} \quad q \in \hat{\boldsymbol{c}} \quad \text{and} \quad \Omega_{l+1}^q = \bigcup_{\hat{q} \in \hat{\boldsymbol{c}}} \Omega_l^{\hat{q}}$$

The condition implies two aspects. First, the communicator of a coarse grid is a subset of or equal to the communicator of the next finer grid. In particular, the number of processes cannot increase on coarser levels, and each active process of a coarse grid is also active on the next finer grid. An important consequence is that processes which are once deactivated on a certain grid level, will stay inactive on all coarser grids. Second, the condition ensures that any subdomain on a coarse grid coincides exactly with the union of possibly several subdomains on the next finer grid, and that the owner process of the coarse subdomain is part of that union. This reduces the necessary communication for a prolongation or restriction to a minimum. For applying the prolongation or restriction operator, any process of a fine grid needs to exchange data with only one process of the next coarser grid. This yields independent local communication patterns between fine and coarse grid processes.

The actual adjustment of the hardware activity is done by pausing and reactivating MPI processes, which causes the CPU cores to enter sleeping C-states during the pause phase. Whenever the multigrid cycle in Algorithm 2 reaches the restriction in step 6, each MPI process checks whether it is involved in the next coarser level. Any process that is not involved in the next coarser level calls the Linux system command `pause` [1] after it contributed to the local communication required for the restriction. When the multigrid cycle returns to this grid from the recursion in step 9, the active processes reactivate all sleeping processes which are involved in this grid by sending them the system signal `SIGUSR1` through the Linux system call `kill` [1].

2.5 Test platforms

We used two different platforms for our numerical experiments. The first platform is a single compute node equipped with four Intel Xeon E-4650 8-core CPUs, where the `pmlib` framework [5] with an external power meter was installed for energy measurements. The power measurement setup is presented in more detail in the next subsection. We used this single node machine for time and energy measurements, running tests with up to all the available 32 CPU cores. The second platform is the HPC cluster system `bwUniCluster` located at the Steinbuch Center for Computing at Karlsruhe Institute of Technology, Germany. The maximum possible allocation was 256 CPU cores. We used this cluster platform to investigate time to solution for a larger number of processes than on the single node.

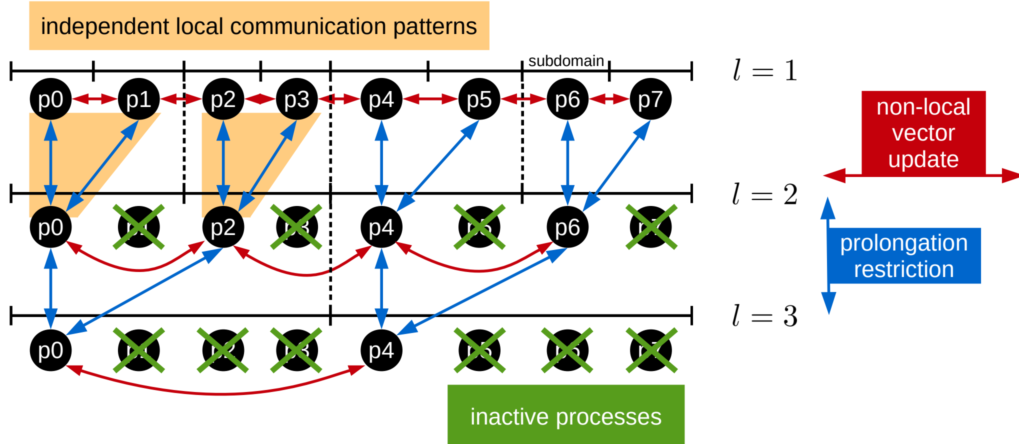

Figure 3: Hierarchy with 3 grids and 8 processes. All processes are active on the fine grid. Every second process is set to sleep on the middle grid, and again every second of the remaining processes is set to sleep on the coarse grid. Non-local vector updates affect only processes of neighboring subdomains. Prolongation and restriction use independent local communication patterns.

2.6 Node-level power measurement

For power measurement on the single node platform, we used the ZES Zimmer Electronic Systems LMG450 external power meter. The node comprised two power supply units, each connected with one line to the external power source. The LMG450 has four independent measurement channels. We used one channel for each of the two input lines, and the other two channels were left unused. We attached the power sensors of the LMG450 to the input lines between the external power source and the power supply units of the compute 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 [5]. 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 overhead 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 4.

3 Results

We performed test runs with varying parallel configurations as well as varying problem size. We investigated five different parallel configurations, as defined in Tab. 1. In the default configuration, all of the CPU cores used for a specific test run are active throughout the whole multigrid hierarchy. To assess the effect of changing the hardware activity during runtime, we used four adapted configurations (A-D) with an increasing number of deactivated cores on coarser grid levels. For the adapted configuration (A), half of the cores are deactivated on the coarsest grid. In the adapted configuration (B), half of the cores are already deactivated on the second coarsest grid, and again half of the remaining cores are deactivated on the coarsest grid. In the adapted configurations (C) and (D) this cascade of deactivations even starts on finer levels. In any case at least one process stayed active. These five parallel configurations were tested with three different problem sizes, as defined in Tab. 2. On the single node platform, we ran tests using $p = 1, 2, 4, 8, 16$ and 32 processes for each parallel configuration and each problem size. For each individual test case we measured time and energy to solution using the platform-independent high resolution `MPI_Wtime` function [3] and the `pmlib` framework as described above in Subsec. 2.6. To guarantee consistency and to prevent from race conditions in the measurements of the parallel application, we encapsulated all timing and `pmlib` calls in globally synchronized blocks using barriers on the `MPI_COMM_WORLD` communicator. Every measurement was repeated five times.

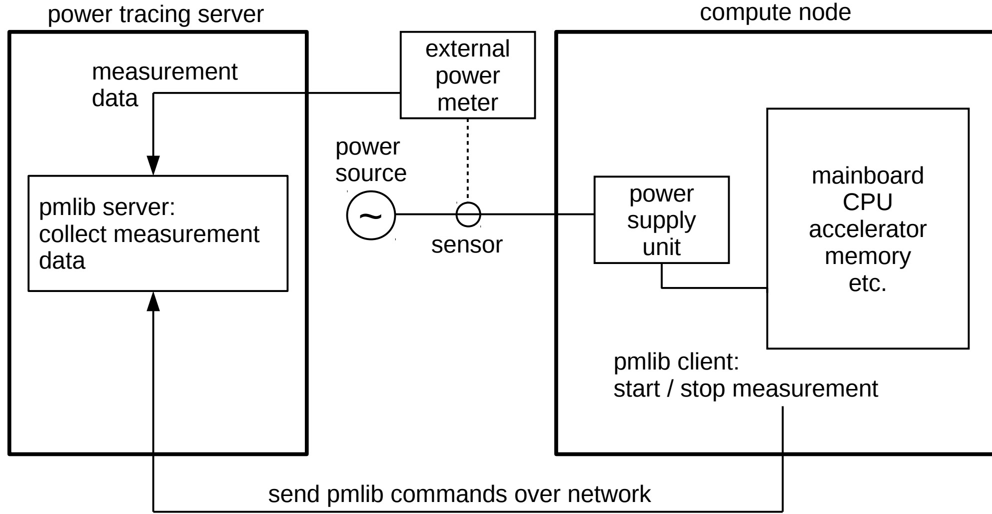

Figure 4: Measurement setup on the single node platform with an external power meter controlled by the pmlib tool.

Additional tests for the medium and large problem sizes with a greater number of processes up to $p = 256$ were carried out on the `bwUniCluster`. Since no energy measurement devices were available on this machine, only time measurements could be taken.

parallel configuration	default	(A)	(B)	(C)	(D)
lvl 1	p	p	p	p	p
lvl 2	p	p	p	p	$p/2$
lvl 3	p	p	p	$p/2$	$p/4$
lvl 4	p	p	$p/2$	$p/4$	$p/8$
lvl 5	p	$p/2$	$p/4$	$p/8$	$p/16$

Table 1: Number of active CPU cores per grid level depending on the number p of cores used for a specific test run. The default parallel configuration uses all cores on all levels, while the adapted configurations (A-D) deactivate cores on coarser levels.

problem size	small	medium	large
n_h	263,169	1,050,625	4,198,401
n_{2h}	66,049	263,169	1,050,625
n_{4h}	16,641	66,049	263,169
n_{8h}	4,225	16,641	66,049
n_{16h}	1,089	4,225	16,641

Table 2: Number of unknowns per grid level in the multigrid hierarchy for three different problem sizes denoted small, medium and large.

3.1 Node-level time and energy measurements

Figure 5 shows the results for time (left) and energy (right) to solution for the **small problem size** test runs. In the default configuration we observed increasing speedups with increasing number of processes up to the case $p = 16$. This case achieved the best time to solution of $t = 0.179$ seconds within the default configuration tests. Using $p = 32$ processes did not yield further improvement, but rather slowed down the execution.

The test runs in the adapted configurations showed a similar behavior with increasing speedups up to $p = 16$ processes. However, in spite of showing speedups, the adapted configurations yielded higher absolute time to solution for the $p = 2, 4, 8$ cases compared to the default configuration. The $p = 16$ case

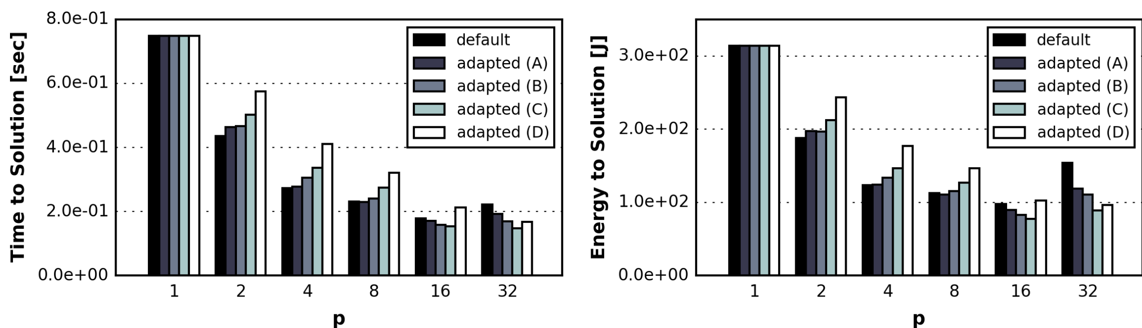

Figure 5: Time to solution (left) and energy to solution (right) for the small problem size test runs on a single compute node.

represents the break-even point for the adapted configurations (A), (B) and (C), which showed smaller time to solution than the default configuration. In the $p = 32$ case all four adapted configurations (A-D) outperformed the default configuration. But increasing the number of processes from 16 to 32 was not beneficial for the (A) and (B) adapted configurations. Only the adapted configurations (C) and (D) showed further speedup for the $p = 32$ case. The overall optimum with respect to time to solution was achieved in the adapted (C) configuration with $t = 0.148$ seconds using $p = 32$ processes. This is an improvement of 17.3% over the best case in the default configuration.

The power measurements yielded the best energy to solution within the default configuration tests for the $p = 16$ case with $E = 97.799$ Joule. Thus, the best time to solution and the best energy to solution cases coincide for the default configuration. In contrast, the overall optimal energy to solution was achieved for the adapted (C) configuration with $E = 77.194$ Joule in the $p = 16$ case, which does not coincide with the optimal time to solution case. The overall optimal energy to solution case saved 21.1% of energy over the best default configuration.

Figure 6 shows the results of the **medium problem size** test runs. Speedups were observed for the

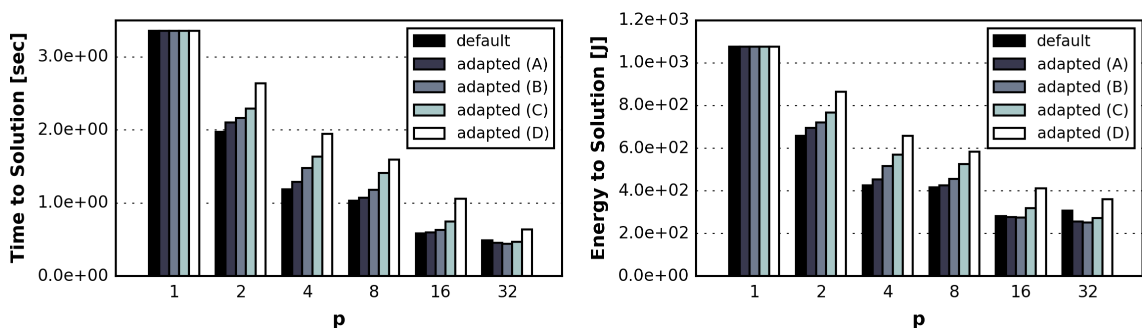

Figure 6: Time to solution (left) and energy to solution (right) for the large medium size test runs on a single compute node.

full range from $p = 1$ to 32 in all parallel configurations. Therefore, the best time to solution within each parallel configuration was always found for $p = 32$ processes. While the best default configuration time to solution was $t = 0.490$ seconds, the overall optimal time to solution resulted from the adapted configuration (B) with $t = 0.443$ seconds, which is a saving of 9.6%. The best default configuration energy to solution was $E = 281.799$ Joule using $p = 16$ processes, whereas the overall optimal energy to solution resulted from the adapted configuration (B) with $E = 251.276$ using $p = 32$ processes. Thus 10.8% of energy could be saved over the best default configuration. Note that for the medium problem size the cases of best time and energy to solution within the default configuration tests do not coincide, while the overall optimal time and energy to solution were achieved in the same adapted configuration (B) using the same number of processes.

The **large problem size** test runs, depicted in Figure 7, showed even more distinct speedups in all

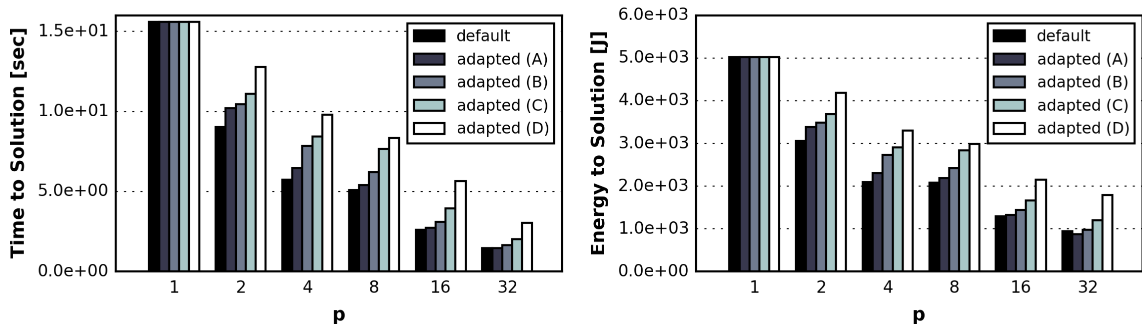

Figure 7: Time to solution (left) and energy to solution (right) for the large problem size test runs on a single compute node.

parallel configurations when varying the number of processes from $p = 1$ to 32. The best time and energy to solution within the default configuration was achieved for the $p = 32$ case with $t = 1.479$ seconds and $E = 948.215$ Joule. Moreover, the default configuration test runs were superior both in terms of time and energy to solution for all cases $p = 1$ to 32 compared to all corresponding adapted configurations, except for one case. Only the adapted configuration (A) case $p = 32$ resulted in an improvement over the default configuration and yielded the overall optimal time and energy to solution with $t = 1.478$ seconds and $E = 868.620$ Joule. Although the difference in the time to solution between the best default case and the overall optimal case is negligible, the energy saving of 8.4% is significant.

Evidence for the effect of pausing certain MPI processes and causing CPU cores to enter deeper C-states,

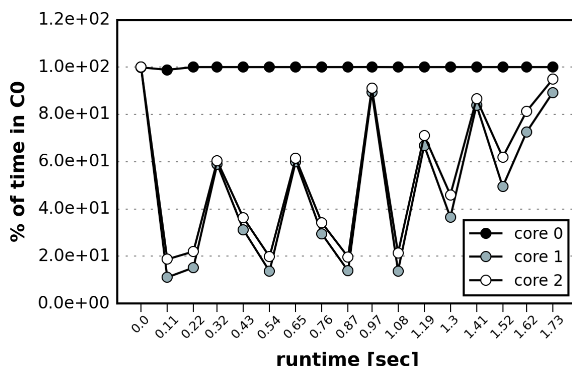

Figure 8: Trace of the C-states for three selected cores during the execution of the multigrid solver in the adapted configuration (B) with 32 processes on the large problem size. The plotted values indicate the percentage of the runtime between consecutive sampling points which the cores spent in C0 state.

thus saving energy, can be seen in Figure 8. It shows the trace of the C-states for three selected cores during the execution of the multigrid solver in the adapted configuration (B) with 32 processes. Core 0 is active throughout the complete solver run, while cores 1 and 2 are deactivated temporarily and thus spent significantly less time in the C0 state. More specifically, every second core is deactivated on the second coarsest grid, which is represented by core 1 in the plot. From the remaining active cores on the second coarsest level, again every second core is deactivated on the coarsest level, which is represented by core 2 in the plot.

3.2 Cluster-level time measurements

For the medium and large problem size, we ran further tests up to $p = 256$ processes on the `bwUniCluster`.

The **medium problem size** test runs, plotted on the left side of Figure 9, showed speedups in the

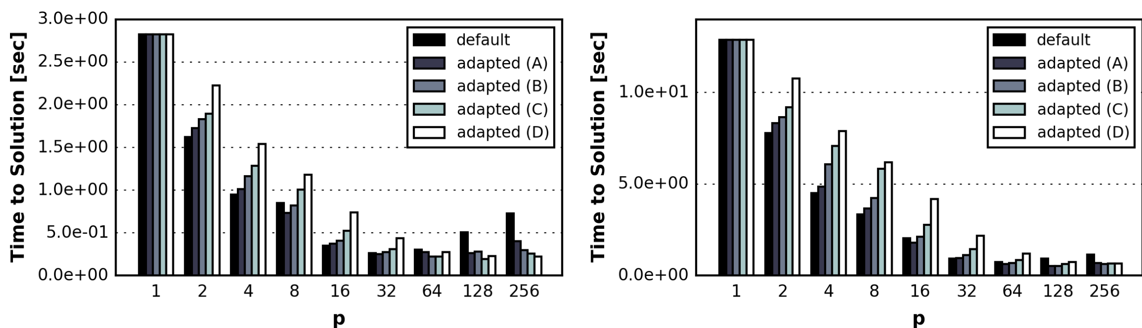

Figure 9: Time to solution on bwUniCluster for the medium problem size (left) and for the large problem size (right).

default configuration up to $p = 32$ processes, as it was also observed in the node-level experiments. However, it turned out that no further speedups were possible for $p = 64, 128$ and 256, but instead the time to solution increased. Consequently, the best default configuration was $p = 32$ as in the node-level tests, with $t = 0.260$ seconds on this machine. The adapted configuration tests were for $p \leq 16$ in most cases inferior to the default configuration. Only the particular cases of the adapted configurations (A) and (B) with $p = 8$ showed slightly better time to solution than the corresponding default configuration run. The break-even point was reached with the transition from $p = 32$ to $p = 64$ processes, where the speedup of the default configuration ceased but the adapted configurations kept improving. The overall optimal time to solution was achieved in the adapted configuration (C) for $p = 128$ with $t = 0.195$ seconds, which was a time saving of 25%.
The right side plot of Figure 9 shows the results of the **large problem size** test runs with speedups in the default configuration up to $p = 64$ processes. Using $p = 128$ and 256 processes slowed down the execution. The best default configuration yielded $t = 0.737$ seconds with $p = 64$. All adapted configurations (A-D) showed speedups up to $p = 128$ processes, and the adapted configuration (D) even showed further speedup with $p = 256$. The break-even point was reached with the transition from $p = 64$ to $p = 128$ processes, where all adapted configuration tests outperformed the default configuration. The overall optimal time to solution was achieved in the adapted configuration (B) for $p = 128$ with $t = 0.526$ seconds, which was a time saving of 28.6%.

4 Conclusion

We investigated the dynamic adjustment of the hardware activity when running a parallel geometric multigrid solver and the effect on runtime performance and energy consumption. Hardware activity was adjusted by means of pausing and reactivating MPI processes, which caused CPU cores to temporarily enter sleeping C-states during the execution of the multigrid algorithm. We presented the multigrid algorithm and our parallelization approach which enables the use of different numbers of MPI processes on different grid levels in the multigrid cycle. We depicted our methodology for time and power measurements of the parallel application on a single compute node with a high precision power meter available, and on an HPC cluster without power measurements. We described our numerical experiments with varying problem sizes to assess the effect of adapting the hardware activity during the solver execution on time and energy to solution. For each problem size which we tested in our experiments, we found adapted hardware activity configurations with temporarily paused MPI processes which outperformed the default configuration both in terms of time and energy to solution. However, the best adaption strategy and the break even point in terms of total number of processes used where the adapted configuration becomes superior over the default configuration, depended on the problem size.
For small problems with a moderate potential for a domain decomposition parallelization, we found that an adaption strategy where processes are not only deactivated on the coarsest level, but already on finer levels, was most beneficial. It resulted in more than 20% of energy savings over the best default case.

Interestingly, the optimal energy to solution case used less processes and longer runtime than the optimal time to solution case, but still consumed less energy due to the temporary deactivation of the hardware. For larger problem sizes, a moderate number of processes deactivated only on the coarsest level and possibly on the second coarsest level turned out to be the best option if only a limited total number of CPU cores is available, as it is often the case for a single compute node. The reason is that the strong scaling of the default configuration is better for larger problem sizes, so that the default parallel configuration's efficiency on the coarsest level does not impair the overall performance as severe as for small problems. Yet the adaption of the hardware activity resulted in slight energy savings around 10%. On the cluster level, the activity adaption strategies showed a much stronger effect. With more CPU cores available, it was beneficial to deactivate cores already on finer levels. This increased the upper limit on the number of processes which are useful for the parallel execution of the multigrid solver in the sense that it yielded further speedup. Remarkable time savings of more than 25% were possible over the best default case.

Acknowledgements

This work was supported by the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 318793.
Parts of this work were performed on the computational resource bwUniCluster funded by the Ministry of Science, Research and the Arts Baden-Württemberg and the Universities of the State of Baden-Württemberg, Germany, within the framework program bwHPC.

References

[1] Linux Manual Section 2: system calls.

[2] BLAS Technical Forum Standard. *International Journal of High Performance Computing Applications*, 2001.

[3] MPI: A Message-Passing Interface Standard, Version 3.0, 2012.

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

[5] 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.

[6] N. Bell, S. Dalton, and L.N.Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM J. Sci. Comput.*, 34:123–152, 2012.

[7] E. Chow, R.D. Falgout, J.J. Hu, R.S. Tuminaro, and U.M. Yang. *Parallel Processing for Scientific Computing*, chapter A Survey of Parallelization Techniques for Multigrid Solvers. SIAM Series on Software, Environments, and Tools. 2006.

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

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

[10] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 1999.

[11] Y. Saad. *Iterative Methods for Sparse Linear Systems.* 2 edition, 2000.

[12] K. Stüben. A review of algebraic multigrid. *J. Comput. Appl. Math.*, 128:281–309, 2001.

[13] P. Wesseling and C.W. Oosterlee. Geometric multigrid with applications to computational fluid dynamics. *J. Comput. Appl. Math.*, 128:311–334, 2001.

[14] M. Wlotzka and V. Heuveline. GPU-accelerated smoothers and grid transfer operators for parallel multigrid methods on HPC clusters. *submitted to: J. Parallel Distrib. Comput.*, Special Issue on Energy Efficient Multi-Core and Many-Core Systems, 2015.

[15] M. Wlotzka and V. Heuveline. Block-asynchronous and Jacobi smoothers for a multigrid solver on GPU-accelerated HPC clusters. *EMCL Prepr. Ser.*, 3, 2015 (to appear).

[16] J. Xu. Iterative Methods by Space Decomposition and Subspace Correction. *SIAM Review*, 34:581–613, 1992.

Preprint Series of the Engineering Mathematics and Computing Lab

recent issues

No. 2017-02 Thomas Loderer, Vincent Heuveline: New sparsing approach for real-time simulations of stiff models on electronic control units

No. 2017-01 Chen Song, Markus Stoll, Kristina Giske, Rolf Bendl, Vincent Heuveline: Sparse Grids for quantifying motion uncertainties in biomechanical models of radiotherapy patients

No. 2016-02 Jonas Kratzke, Vincent Heuveline: An analytically solvable benchmark problem for fluid-structure interaction with uncertain parameters

No. 2016-01 Philipp Gerstner, Michael Schick, Vincent Heuveline, Nico Meyer-Hbner, Michael Suriyah, Thomas Leibfried, Viktor Slednev, Wolf Fichtner, Valentin Bertsch: A Domain Decomposition Approach for Solving Dynamic Optimal Power Flow Problems in Parallel with Application to the German Transmission Grid

No. 2015-04 Philipp Gerstner, Vincent Heuveline, Michael Schick : A Multilevel Domain Decomposition approach for solving time constrained Optimal Power Flow problems

No. 2015-03 Martin Wlotzka, Vincent Heuveline: Block-asynchronous and Jacobi smoothers for a multigrid solver on GPU-accelerated HPC clusters

No. 2015-02 Nicolai Schoch, Fabian Kiler, Markus Stoll, Sandy Engelhardt, Raffaele de Simone, Ivo Wolf, Rolf Bendl, Vincent Heuveline: Comprehensive Pre- & Post-Processing for Numerical Simulations in Cardiac Surgery Assistance

No. 2015-01 Teresa Beck, Martin Baumann, Leonhard Scheck, Vincent Heuveline, Sarah Jones: Comparison of mesh-adaptation criteria for an idealized tropical cyclone problem

No. 2014-02 Christoph Paulus, Stefan Suwelack, Nicolai Schoch, Stefanie Speidel, Rdiger Dillmann, Vincent Heuveline: Simulation of Complex Cuts in Soft Tissue with the Extended Finite Element Method (X-FEM)

No. 2014-01 Martin Wlotzka, Vincent Heuveline: A parallel solution scheme for multiphysics evolution problems using OpenPALM

No. 2013-04 Nicolai Schoch, Stefan Suwelack, Stefanie Speidel, Rediger Dillmann, Vincent Heuveline: Simulation of Surgical Cutting in Soft Tissue using the Extended Finite Element Method (X-FEM)

No. 2013-03 Martin Wlotzka, Edwin Haas, Philipp Kraft, Vincent Heuveline, Steffen Klatt, David Kraus, Klaus Butterbach-Bahl, Lutz Breuer: Dynamic Simulation of Land Management Effects on Soil N2O Emissions using a coupled Hydrology-Ecosystem Model

No. 2013-02 Nicolai Schoch, Stefan Suwelack, Rüediger Dillmann, Vincent Heuveline: Simulation of Surgical Cutting in Soft Tissue using the Extended Finite Element Method (X-FEM)

No. 2013-01 Martin Schindewolf, Björn Rocker, Wolfgang Karl, Vincent Heuveline: Evaluation of two Formulations of the Conjugate Gradients Method with Transactional Memory

No. 2012-07 Andreas Helfrich-Schkarbanenko, Vincent Heuveline, Roman Reiner, Sebastian Ritterbusch: Bandwidth-Efficient Parallel Visualization for Mobile Devices

The responsibility for the contents of the working papers rests with the authors, not the Institute. Since working papers are of a preliminary nature, it may be useful to contact the authors of a particular working paper about results or caveats before referring to, or quoting, a paper. Any comments on working papers should be sent directly to the authors.

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

www.emcl.iwr.uni-heidelberg.de