A Massively Parallel Algebraic Multigrid Preconditioner based on Aggregation for Elliptic Problems with Heterogeneous Coefficients

Article · September 2012

Source: arXiv

CITATIONS
9

READS
133

3 authors, including:

Markus Blatt
Dr. Markus Blatt - HPC-Simulation Software & Services
16 PUBLICATIONS **809** CITATIONS

SEE PROFILE

Olaf Ippisch
Technische Universität Clausthal
82 PUBLICATIONS **884** CITATIONS

SEE PROFILE

A Massively Parallel Algebraic Multigrid Preconditioner based on Aggregation for Elliptic Problems with Heterogeneous Coefficients

Markus Blatt[†*] Olaf Ippisch[†] Peter Bastian[†]

Abstract

This paper describes a massively parallel algebraic multigrid method based on non-smoothed aggregation. It is especially suited for solving heterogeneous elliptic problems as it uses a greedy heuristic algorithm for the aggregation that detects changes in the coefficients and prevents aggregation across them. Using decoupled aggregation on each process with data agglomeration onto fewer processes on the coarse level, it weakly scales well in terms of both total time to solution and time per iteration to nearly 300,000 cores. Because of simple piecewise constant interpolation between the levels, its memory consumption is low and allows solving problems with more than 10^{11} degrees of freedom.

Key words: algebraic multigrid, parallel computing, preconditioning, HPC, high-performance-computing

AMS subject classification: 65F08, 65N08, 65N55, 65Y05

1 Introduction

When solving elliptic or parabolic partial differential equations (PDEs) most of the computation time is often spent in solving the arising linear algebraic equations. This demands for highly scalable parallel solvers capable of running on recent supercomputer. The current trend in the development of high performance supercomputers is to build machines that utilize more and more cores with less memory per core, but interconnected with low latency networks. To be able to still solve problems of reasonable size the parallel linear solvers need to be (weakly) scalable and have a very small memory footprint.

Besides domain decomposition methods the most scalable and fastest methods are multigrid methods. They can solve these linear systems with optimal

*Dr. Markus Blatt - HPC-Simulation-Software & Services, Hans-Bunte-Str. 8-10, D-69123 Heidelberg, email: markus@dr-blatt.de

†Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Ruprechts-Karls-Universität Heidelberg, Im Neuenheimer Feld 368, D-69120 Heidelberg, Germany

or nearly optimal complexity, i.e. at most $O(N \log N)$ operations for N unknowns. Among them algebraic multigrid methods (AMG) are especially suited for problems with heterogeneous or anisotropic coefficient tensors on unstructured grids. They build a hierarchy of matrices using their graphs and thus adapt the coarsening to the problem solved.

Parallel geometric multi-grid implementations exist since at least 25 years, see e.g. [16]. Since about 15 years, several parallel algebraic multigrid codes have been developed [15, 29, 20, 3, 18]. Classical AMG [26] divides the fine level unknowns into two groups: the ones also represented on the coarse level, and the ones that exist only on the fine level. In its parallel version the splitting of the unknowns has to be globally consistent. This makes the coarsening of unknowns, that are either also represented on other processes or adjacent to unknowns on them, not only difficult but also inherently sequential near the inter-process boundary. It causes more communication and increases operator complexity (sum of the number of nonzeros of the matrices on all levels divided by the number of nonzeros of the fine level matrix). Therefore, especially in 3D, the time needed for building the operator hierarchy and the time needed to perform one iteration increase with the problem size and the number of cores used, albeit the number of iterations needed for convergence stays optimal. Adapted coarsening heuristics and aggressive coarsening strategies have been developed to partly overcome this problem [28, 13, 1].

AMG based on aggregation, see [12, 30, 25], clusters the fine level unknowns into aggregates. Each aggregate represents an unknown on the coarse level and its basis function is a linear combination of the fine level basis functions associated with the aggregate. Two main classes of the method exist. Non-smoothed aggregation AMG, see [12, 25, 22, 8], which uses simple piecewise constant interpolation, and smoothed aggregation AMG, that increases interpolation accuracy by smoothing the tentative piecewise constant interpolation. For the parallel versions of both classes no growth in operator complexity is observed for increasing numbers of processes [29, 22, 8]. Still the smoothing of the interpolation operators increases the stencil size of the coarse level matrices compared to the non-smoothed version. Moreover, the non-smoothed version can be used in straight forward way for many systems of PDEs, see [8].

In this paper we describe a parallel AMG method that uses a greedy heuristic algorithm for the aggregation based on a strength of connection criterion. This allows for building round aggregates of nearly arbitrary size that do not cross high contrast coefficient jumps. We use simple piecewise constant interpolation between the levels to prevent an increase of the size of the coarse level stencils. Together with an implementation of the parallel linear algebra operations based on index sets this makes the algorithm very scalable regarding the time needed per iteration. Even though the number of iterations needed for convergence do increase during weak scalability tests, the time to solution is still very scalable. We present numerical evidence that the approach is scalable up to 262,144 cores for realistic problems with highly variable coefficients. At the same time the memory requirement of the algorithm is far less than that of classical AMG methods. This allows us to solve problems with more than 10^{11} degrees of

unknowns on an IBM Blue Gene/P using 64 racks.

We will start the paper in the next section with a description of the algebraic multigrid method together with our heuristic greedy aggregation algorithm for coarsening the linear systems. In Section 3 we describe the parallelization of the algebraic multigrid solver and its components, namely the data decomposition, smoothers, interpolation operators, and linear operators. After presenting implementational details about the parallelization and linear algebra data structures in Section 4, we conduct scalability tests of our method on an IBM Blue Gene/P and an off-the-shelf multicore Linux cluster in Section 5. Our summary and conclusions can be found in Section 6.

2 Algebraic Multigrid

For any finite index set $I \subset \mathbb{N}$ we define the vector space $\mathbb{R}^I$ to be isomorphic to $\mathbb{R}^{|I|}$ with components indexed by $i \in I$. Thus $\mathbf{x} \in \mathbb{R}^I$ can be interpreted as a mapping $\mathbf{x} : I \to \mathbb{R}$ and $x_i = \mathbf{x}(i)$. In the same way, for any two finite index sets $I, J \subset \mathbb{N}$ we write $\mathsf{A} \in \mathbb{R}^{I \times J}$ with the interpretation $\mathsf{A} : I \times J \to \mathbb{R}$ and $a_{i,j} = \mathsf{A}(i,j)$. Finally, for any subset $I' \subseteq I$ we define the restriction matrix $\mathsf{R}_{I,I'} : \mathbb{R}^I \to \mathbb{R}^{I'}$ as $(\mathsf{R}_{I,I'}\mathbf{x})_i = x_i \ \forall i \in I'$ (which corresponds to simple injection).

On a given domain Ω we are interested in solving the model problem

$$\nabla \cdot (\mathsf{K}(x)\nabla u) = f\,, \quad \text{on } \Omega \tag{1}$$

together with appropriate boundary conditions. Here, the symmetric positive definite tensor $\mathsf{K}(x)$, dependent on the position x within the domain Ω, is allowed to be discontinuous. Given an admissible mesh T_h that for simplicity resolves the boundary and possible discontinuities in the tensor $\mathsf{K}(x)$, discretizing (1) using conforming lowest order Galerkin finite element or finite volume methods yields a linear system

$$\mathsf{A}x = b\,, \tag{2}$$

where $\mathsf{A} : \mathbb{R}^n \to \mathbb{R}^n$ is the linear operator, and $x, b \in \mathbb{R}^n$ are vectors. For an extension to discontinuous Galerkin methods see [6]. We strive to solve this linear system using our algebraic multigrid method described below.

The excellent computational complexity of multgrid methods is due to the following main idea. Applying a few steps of a smoother (such as Jacobi or Gauss-Seidel) to the linear system usually leads to a smooth error that cannot be reduced well by further smoothing. Given a prolongation operator P from a coarser linear system, this error is then further reduced using a correction u^{coarser} on a coarser linear system $\mathsf{P}^T\mathsf{A}\mathsf{P}\mathbf{u}^{\text{coarse}} = \mathsf{P}^T(\mathbf{b} - \mathsf{A}\mathbf{x})$. We use the heuristic algorithm presented in Subsection 2.1 to build the prolongation operator P. If the system is already small enough, we solve it using a direct solver. Otherwise we recursively apply a few steps of the smoother and proceed to an even coarser linear system until the size of the coarsest level is suitable for a direct solver. After applying the coarse level solver, we prolongate the correction to the next finer level, add it to the current guess, and apply a few steps of the smoother.

2.1 Coarsening by Aggregation

To define the prolongation operator P we rely on a greedy and heuristic aggregation algorithm, that uses the graph of the matrix as input. It is an extension of the version published by Raw (cf. [24]) for algebraic multigrid methods (see also [27]).

Let $G = (\mathcal{V}, \mathcal{E})$ be a graph with a set of vertices $\mathcal{V}$ and edges $\mathcal{E}$ and let $w_{\mathcal{E}} : \mathcal{E} \to \mathbb{R}$ and $w_{\mathcal{V}} : \mathcal{V} \to \mathbb{R}$ be positive weight functions. For the examples in this paper $w_{\mathcal{E}}((i,j)) = \frac{1}{2}(a_{ij} - |\, a_{ij}|)$ is used, i.e. 0 for positive off-diagonals and the absolute value otherwise, and $w_{\mathcal{V}}(i) = a_{ii}$. For matrices arising from the discretisation of systems of PDE, for which our aggregation scheme is applicable as well, $w_{\mathcal{E}}$ and $w_{\mathcal{V}}$ could e.g. be the row-sum norm of a matrix block (see e.g. [8]). These functions are used to classify the edges and vertices of our graph. Let

$$N(i) := \{j \in \mathcal{V} \mid \exists (j,i) \in \mathcal{E}\}$$

be the set of adjacent vertices of vertex i and let

$$\eta_{\max}(i) := \max_{k \in N(i)} \frac{w_{\mathcal{E}}((k,i))\, w_{\mathcal{E}}((i,k))}{w_{\mathcal{V}}(i)\, w_{\mathcal{V}}(k)}. \tag{3}$$

(a) An edge (j,i) is called *strong*, if and only if

$$\frac{w_{\mathcal{E}}((i,j))\, w_{\mathcal{E}}((j,i))}{w_{\mathcal{V}}(i)\, w_{\mathcal{V}}(j)} > \delta\ \min(\eta_{\max}(i), \eta_{\max}(j)), \tag{4}$$

for a given threshold $0 < \delta < 1$. We denote by $N_\delta(i) \subset N(i)$ the set of all vertices adjacent to i that are connected to it via a strong edge.

(b) A vertex i is called *isolated* if and only if $\eta_{\max}(i) < \beta$, for a prescribed threshold $0 < \beta \ll 1$. We denote by $ISO(\mathcal{V}) \subset \mathcal{V}$ the set of all isolated vertices of the graph.

For symmetric positive definite M-matrices arising from problems with constant diffusion coefficients our strength of connection criterion is similar to the traditional ones for the AMG of Ruge and Stüben [26] and for smoothed aggregation [31]. For non-symmetric matrices or problems with discontinuous coefficients it differs from them. It is especially tailored for the latter. At the interfaces of the jumps the Ruge Stüben criterion might classify a connection between two vertices strong in one direction and weak in the other one. Actually, no aggregation should happen across this interface. The smoothed aggregation criterion falsely classifies positive off-diagonal values as strong while ours does not do this with an appropriate weight function. For more details see [8].

Our greedy aggregation algorithm is described in Algorithm 1.

Algorithm 1 Build Aggregates

procedure AGGREGATION($\mathcal{V}$, $\mathcal{E}$, $s_{\min}$, $s_{\max}$, $d_{\max}$)
  $U \leftarrow \mathcal{V} \setminus ISO(\mathcal{V})$ ▷ First candidates are non-isolated vertices
  $I \leftarrow \emptyset$ ▷ Coarse index set
  $S \leftarrow \{u \in U : \#N_{\text{na}}(u) \leq \#N_{\text{na}}(w)\ \forall w \in U\}$ ▷ Seed stack
  Pop seed v from S
  while $U \neq \emptyset$ **do**
    $a_v \leftarrow \{v\}$ ▷ Initialize new aggregate
    $U \leftarrow U \setminus a_v$
    $I \leftarrow I \cup \{v\}$
    GROWAGGREGATE(a_v, $\mathcal{V}$, $\mathcal{E}$, $s_{\min}$, $d_{\max}$, U)
    ROUNDAGGREGATE(a_v, $\mathcal{V}$, $\mathcal{E}$, $s_{\max}$, U)
    if $\#a_v = 1$ **then** ▷ Merge one vertex aggregate with neighbor
      $C \leftarrow \{a_j : j \in I \setminus \{v\}$ and $\exists w \in a_j$ with $w \in N_\delta(v)\}$
      if $C \neq \emptyset$ **then**
        Choose $a_k \in C$
        $I \leftarrow I \setminus \{v\}$
        $a_k \leftarrow a_k \cup a_v$
      end if
    end if
    $S \leftarrow \{w : w \in N(a_v)\}$
    if $U \neq \emptyset$ **then**
      if $S = \emptyset$ **then**
        $S \leftarrow \{u \in U : \#N_{\text{na}}(u) \leq \#N_{\text{na}}(w)\ \forall w \in U\}$
      end if
      Pop seed v from S
    end if
  end while
  $U \leftarrow ISO(\mathcal{V})$ ▷ Aggregate isolated vertices
  while $U \neq \emptyset$ **do**
    Select arbitrary seed $v \in U$
    $a_v \leftarrow \{v\}$
    $U \leftarrow U \setminus a_v$
    $I \leftarrow I \cup \{v\}$
    GROWISOAGGREGATE(a, $\mathcal{V}$, $\mathcal{E}$, $s_{\min}$, $d_{\max}$, U)
  end while
  $\mathcal{A} \leftarrow \{a_i : i \in I\}$
  return $(\mathcal{A}, I)$
end procedure

Until all non-isolated vertices are aggregated, we start a new aggregate a_v with a non-isolated vertex v. To select this vertex we use a finite stack. Whenever we try to pop a vertex from an empty stack, we fill it with the vertices, which have the least number of non-aggregated neighbors N_{na}. Each time an aggregate is finished, all non-isolated non-aggregated neighbors are pushed onto

the stack. The index of the seed vertex is associated with this new aggregate and added to the index set I. The algorithm returns both the index set I for the set of aggregates as well as the set $\mathcal{A} = \{a_i : i \in I\}$ of all aggregates it has built.

The first step in the construction of an aggregate in Algorithm 1 is to add new vertices to our aggregate until we reach the minimal prescribed aggregate size $s_{\min}$. This is outlined in Algorithm 2.

Algorithm 2 Grow Aggregate Step

function GROWAGGREGATE(a, $\mathcal{V}$, $\mathcal{E}$, $s_{\min}$, $d_{\max}$, U)
  while $\#a \leq s_{\min}$ **do** ▷ Makes aggregate a bigger until its size is $s_{\min}$
    $C_0 \leftarrow \{v \in N(a) : \ \mathrm{diam}(a, v) \leq d_{\max}\}$ ▷ Limit the diameter of the aggregate
    $C_1 \leftarrow \{v \in C_0 : \ \mathrm{cons}_2(v, a) \geq \ \mathrm{cons}_2(w, a) \ \forall w \in N(a)\}$
    if $C_1 = \emptyset$ **then** ▷ No candidate with two-way connections
      $C_1 \leftarrow \{v \in C_0 : \ \mathrm{cons}_1(v, a) \geq \ \mathrm{cons}_1(w, a) \ \forall w \in N(a)\}$
    end if
    if $\#C_1 > 1$ **then** ▷ More than one candidate
      $C_1 \leftarrow \{v \in C_1 : \frac{\mathrm{connect}(v,a)}{\#N(v)} \geq \frac{\mathrm{connect}(w,a)}{\#N(w)} \ \forall w \in C_1\}$
    end if
    if $\#C_1 > 1$ **then** ▷ More than one candidate
      $C_1 \leftarrow \{v \in C_1 : \ \mathrm{neighbors}(v, a) \geq \ \mathrm{neighbors}(w, a) \ \forall w \in C_1\}$
    end if
    if $C_1 = \emptyset$ **then break**
    end if
    Select one candidate $c \in C_1$
    $a \leftarrow a \cup \{c\}$ ▷ Add candidate to aggregate
    $U \leftarrow U \setminus \{c\}$
  end while
end function

When adding new vertices, we always choose a vertex within the prescribed maximum diameter $d_{\max}$ of the aggregate which has the highest number of strong connections to the vertices already in the aggregate. Here we give preference to vertices where both edge (i, j) and edge (j, i) are strong. The functions $\mathrm{cons}_1(v, a)$ and $\mathrm{cons}_2(v, a)$ return the number of one-way and two-way strong connections between the vertex v and all vertices of the aggregate a, respectively.

If there is more than one candidate, we want to choose a vertex with a high proportion of strong connections to other vertices not belonging to the current aggregate, while favoring connections to vertices which belong to aggregates that are already connected to the current aggregate. We therefore define a function $\mathrm{connect}(v, a)$, which counts neighbors of v that are not yet aggregated or belong to an aggregate that is not yet connected to aggregate a once and neighbors of v that belong to aggregates that are already connected to aggregate a twice.

If there is still more than one candidate which maximizes $\frac{\mathrm{connect}(v,a)}{\#N(v)}$, we

choose the candidate, which has the maximal number neighbors(v, a) of neighbors of vertex v that are not yet aggregated neighbors of the aggregate a. This criterion tries to maximize the number of candidates for choosing the next vertex.

Algorithm 3 Round Aggregate Step

function RoundAggregate(a, $\mathcal{V}$, $\mathcal{E}$, $s_{\max}$, U)
  while $\#a \leq s_{\max}$ **do** ▷ Rounds aggregate a while size $< s_{\max}$
    $D \leftarrow \{w \in N(v) \cap U : \ \text{cons}_2(w, a) > 0 \text{ or } \text{cons}_1(w, a)\} > 0$
    $C \leftarrow \{v \in D : \#\{N(v) \cap U\} > \#\{N(v) \cap U\}\}$
    Select arbitrary candidate $c \in C$
    $a \leftarrow a \cup \{c\}$ ▷ Add candidate to aggregate
    $U \leftarrow U \setminus \{c\}$
  end while
end function

In a second step we aim to make the aggregates "rounder". This is sketched in Algorithm 3. We add all non-aggregated adjacent vertices that have more connections to the current aggregate than to other non-aggregated vertices until we reach the maximum allowed size $s_{\max}$ of our aggregate. The function $\text{cons}(v, C)$ returns the number of strong connections between vertex v and the members of the set C.

If after these two steps an aggregate still consists of only one vertex, we try to find another aggregate that the vertex is strongly connected to. If such an aggregate exists, we add the vertex to that aggregate and choose a new seed vertex.

Finally, once all the non-isolated vertices are aggregated, we try to build aggregates for the isolated vertices. Where possible, we build these by aggregating adjacent isolated vertices that have at least one common neighboring aggregate consisting of non-isolated vertices. This is done in the function GrowIsoAggregate which we do not present here. The most common situation where connected isolated vertices occur is when Dirichlet boundary conditions are represented as unknowns in linear systems. In this case their aggregation does not introduce any errors, but prevents the stencils of the coarse level matrices from filling up and thus leads to reasonable coarsening rates and operator complexities. Other situations of such occurrences are very rare.

Given the aggregate information $\mathcal{A}$, we define the piecewise constant prolongation operator P by

$$\mathsf{P}(i, j) = \begin{cases} 1 & \text{if } j \in a_i \\ 0 & \text{else} \end{cases} \tag{5}$$

and define the coarse level matrix using a Galerkin product as

$$\mathsf{A}^{\text{coarse}} = \frac{1}{\omega}\mathsf{P}^T\mathsf{A}\mathsf{P}\,.$$

Note that the overrelaxation factor ω is needed to improve the approximation

properties of the coarse correction. According to [12] $\omega = 1.6$ is a good default and often sufficient for good convergence.

3 Parallelization

3.1 Data Decomposition and Local Data Structures

The most important and computationally expensive part in both algebraic multigrid methods and many iterative solvers (especially Krylov methods and stationary iterative methods) is the application of linear operators. Therefore their construction is crucial. It has to be made in a way that allows for efficient application of the linear operator as well as the construction of preconditioners or smoothers from these linear operators.

Let $I \subset \mathbb{N}$ be our finite index set and $\bigcup_{p\in\mathcal{P}} I_{(p)}$ be a (non-overlapping) partitioning of the index set between the processes $\mathcal{P}$ participating in the computation. This partitioning might be given by an external partitioning software or by a parallel grid manager used for e.g. a parallel finite element discretization. In some cases these tools may not provide a partitioning but an overlapping decomposition. In this case one can easily compute a partitioning following [10].

Unfortunately, using such a partitioning and storing on process p only the subset of entries of the global vector and rows of the global matrix that is associated with its part of the index set $I_{(p)}$, would mean triggering one communication for each matrix entry a_{ij}, with $i \in I_{(p)}$ and $j \notin I_{(p)}$. Our goal is to separate the communication from the application of the linear operator. Therefore each process p stores data associated with a larger index set

$$\widetilde{I}_{(p)} = I_{(p)} \cup \{j \in I | \exists i \in I_{(p)} \,:\, a_{ij} \neq 0\},$$

namely $\widetilde{\mathsf{A}}_{(p)} \in \mathbb{R}^{\widetilde{I}_{(p)}\times\widetilde{I}_{(p)}}$ and $\widetilde{\mathbf{x}}_{(p)}, \widetilde{\mathbf{b}}_{(p)} \in \mathbb{R}^{\widetilde{I}_{(p)}}$. We call $\widetilde{\mathbf{x}}_{(p)}$ stored consistently if $\widetilde{\mathbf{x}}_{(p)} = \mathsf{R}_{I,\widetilde{I}_{(p)}}\mathbf{x}$ for all $p \in \mathcal{P}$. If $\widetilde{\mathbf{x}}_{(p)} = \mathsf{R}^T_{\widetilde{I}_{(p)},I_{(p)}}\mathsf{R}_{I,I_{(p)}}\mathbf{x}$ holds for all $p \in \mathcal{P}$, we denote $\widetilde{\mathbf{x}}_{(p)}$ as uniquely stored.

Let $\mathsf{A} \in \mathbb{R}^{I\times I}$ be a global linear operator. Then on process p the local linear operator $\widetilde{\mathsf{A}}_{(p)}$ stores the values

$$\widetilde{\mathsf{A}}_{(p)}(i,j) = \begin{cases} \mathsf{A}(i,j) & \text{if } i \in I_{(p)} \\ \delta_{i,j} & \text{else} \end{cases} \tag{6}$$

where $\delta_{i,j}$ denotes the usual Kronecker delta. Denoting $\mathsf{A}_{(p)} = \mathsf{R}_{I,I_{(p)}}\mathsf{A}\mathsf{R}^T_{I,I_{(p)}}$ and reordering the indices locally, such that $i < j$ for all $i \in \widetilde{I}_{(p)}$ and $j \notin \widetilde{I}_{(p)} \setminus I_{(p)}$, $\widetilde{\mathsf{A}}_{(p)}$ has the following structure:

$$\begin{array}{l} \widetilde{I}_{(p)} \left\{ \begin{array}{l} I_{(p)} \left\{ \right. \\ \\ \end{array} \right. \end{array} \begin{array}{|c|c|} \hline \mathsf{A}_{(p)} & * \\ \hline 0 & I \\ \hline \end{array} .$$

Using this storage scheme for the local linear operator $\widetilde{\mathsf{A}}_{(p)}$ and applying it to a local vector $\widetilde{\mathbf{x}}_{(p)}$, stored consistently, $(\widetilde{\mathsf{A}}_{(p)}\widetilde{\mathbf{x}}_{(p)})(i) = (\mathsf{A}\mathbf{x})(i)$ holds for all $i \in I_{(p)}$. Therefore the global application of the linear operator can be represented by computing

$$\mathsf{A}\mathbf{x} = \sum_{p\in\mathcal{P}} R^T_{I,I_{(p)}} R_{\widetilde{I}_{(p)},I_{(p)}} \left(\widetilde{\mathsf{A}}_{(p)} R_{I,\widetilde{I}_{(p)}} \mathbf{x}\right) . \tag{7}$$

Here the operators in front of the brackets represent a restriction of the results of the local computation to the (consistent) representation on $\mathbb{R}^{I_{(p)}}$ then a prolongation to global representations and a summation of all these global representations. In contrast to the notation used there is no global summation and thus no global communication needed. It suffices that every process adds only entries from other processes that actually store data associated with indices in $\widetilde{I}_{(p)}$. Therefore this represents a next neighbor communication followed by a local summation.

3.2 Parallel Smoothers

As smoothers we only consider so-called hybrid smoothers [21]. These can be seen as block-Jacobi smoothers where the blocks are the matrices $A_{(p)}$. Instead of directly solving the block systems a few steps of a sequential smoother (e.g. Gauss-Seidel for hybrid Gauss-Seidel) is applied. We always use only one step.

Let $\mathsf{M}_{(p)} \in \mathbb{R}^{I_{(p)}\times I_{(p)}}$, $p \in \mathcal{P}$, be the sequential smoother computed for matrix $\mathsf{A}_{(p)}$, and $\widetilde{\mathbf{d}}_{(p)}$ the defect. Then the consistently stored update $\widetilde{\mathbf{v}}_{(p)}$ is computed by applying the parallel preconditioner as

$$\widetilde{\mathbf{v}}_{(p)} = \mathsf{R}_{I,\widetilde{I}_{(p)}} \sum_{p\in\mathcal{P}} \mathsf{R}^T_{I,I_{(p)}} \left(\mathsf{M}_{(p)} R_{\widetilde{I}_{(p)},I_{(p)}} \widetilde{d}_{(p)}\right) .$$

Again as in the parallel linear operator 7 the summation requires only a communication with processes, which share data associated with $\widetilde{I}_{(p)}$, and thus can be handled very efficiently.

3.3 Parallel Coarsening

The parallelization of the coarsening algorithm described in Section 2.1 is rather straightforward. It is simple and massively parallel since the aggregation only occurs on vertices of the graph of matrix $A_{(p)}$. Using this approach, the coarsening process will of course deal better with the algebraic smoothness if the disjoint matrix $A_{(p)}$ is split along weak edges.

The parallel approach is described in Algorithm 4. It builds the aggregates $\widetilde{\mathcal{A}}_{(p)}$ of this level and the parallel index sets $\widetilde{I}^{\text{coarse}}_{(p)}$ for the next level in parallel. The parameters are the edges and vertices of the matrix graph $G(\widetilde{\mathsf{A}}_{(p)}) = (\widetilde{I}_{(p)}, \widetilde{\mathcal{E}}_{(p)})$ and the disjoint index set $I_{(p)}$. The rest of the parameters are the same as for the sequential Algorithm 1. As a first step a subset

$(I_{(p)}, \mathcal{E}_{(p)})$ of the input graph that corresponds to the index set $I_{(p)}$ is created. Then the sequential aggregation algorithm is executed on this sub-graph. Based on the outcome of this aggregation a map between indices and corresponding aggregate indices is built and the information is published to all other processes that share vertices of the overlapping graph. Now every process knows the aggregate index of each vertex of its part of the overlapping graph and constructs the overlapping coarse index set and the aggregates. Note that this algorithm only needs one communication step per level with the direct neighbors.

Algorithm 4 Parallel Aggregation

procedure ParallelAggregation($I_{(p)}$, $\widetilde{\mathcal{E}}_{(p)}$, $s_{\min}$, $s_{\max}$, $d_{\max}$)
  On process $p \in \mathcal{P}$**:**
  $\mathcal{E}_{(p)} \leftarrow \{(k,l) \in \widetilde{\mathcal{E}}_{(p)} \mid k \in I_{(p)} \wedge l \in I_{(p)}\}$ ▷ Only edges within $I_{(p)}$
  $(I^{\text{coarse}}_{(p)}, \mathcal{A}_{(p)}) \leftarrow$Aggregation($I_{(p)}$, $\mathcal{E}_{(p)}$, $s_{\min}$, $s_{\max}$, $d_{\max}$)
  $\widetilde{\mathbf{m}}_{(p)} \leftarrow 0 \in \mathbb{N}^{\widetilde{I}_{(p)}}$
  for $a_k \in \mathcal{A}_{(p)}$ **do**
    $(\mathsf{R}^T_{I,\widetilde{I}_{(p)}} \widetilde{\mathbf{m}}_{(p)})_j \leftarrow k \quad \forall v_j \in a_k$
  end for
  $\widetilde{\mathbf{m}}_{(p)} \leftarrow R_{I,\widetilde{I}_{(p)}} \sum_{q \in \mathcal{P}} \mathsf{R}^T_{I,\widetilde{I}_{(q)}} \widetilde{\mathbf{m}}_{(q)}$ ▷ Communicate aggregates mapping
  $\widetilde{I}^{\text{coarse}}_{(p)} \leftarrow \{k \mid \exists j \in \widetilde{I}_{(p)} \text{ with } (R^T_{I,\widetilde{I}_{(p)}} \widetilde{\mathbf{m}}_{(p)})_j = k\}$ ▷ Build coarse index set
  for $k \in \widetilde{I}^{\text{coarse}}_{(p)}$ **do**
    $a_k \leftarrow \{j \in \widetilde{I}_{(p)} \mid (R^T_{I,\widetilde{I}_{(p)}} \widetilde{\mathbf{a}}_{(p)})_j = k\}$
  end for
  $\widetilde{\mathcal{A}}_{(p)} \leftarrow \{a_k : k \in \widetilde{I}^{\text{coarse}}_{(p)}\}$ ▷ Aggregate information
  return $(\widetilde{I}^{\text{coarse}}_{(p)}, \widetilde{\mathcal{A}}_{(p)})$
end procedure

For each aggregate a_k on process p, that consists of indices in $\widetilde{I}_{(p)} \setminus I_{(p)}$ on the fine level, the child node, representing that aggregate on the next coarser level, is again associated with an index $i \in a_k \subset \widetilde{I}_{(p)} \setminus I_{(p)}$. This means that for all vertices in $I_{(p)}$ on the coarse level all neighbors they depend on or influence are also stored in process p.

The local prolongation operator $\widetilde{P}_{(p)}$ is calculated from the aggregate information $\widetilde{\mathcal{A}}_{(p)}$ in accordance to (5). Let $\widetilde{\mathsf{A}}^l_{(p)}$ be the local fine level matrix, then the tentative coarse level matrix is computed by the Galerkin product $\widetilde{\mathsf{A}}^{l+1}_{(p)} = \widetilde{\mathsf{P}}^T_{(p)} \widetilde{\mathsf{A}}^l_{(p)} \widetilde{\mathsf{P}}_{(p)}$. To satisfy the constraints of our local operators (6), we need to set the diagonal values to 1 and the off-diagonal values to 0 for all matrix rows corresponding to the overlap region $\widetilde{I}^{l+1}_{(p)} \setminus I^{l+1}_{(p)}$. Due to the structure of the matrices in the hierarchy all matrix-vector operations can be performed locally on each processor provided that the vectors are stored consistently.

3.3.1 Agglomeration on Coarse Levels

Note that our aggregation Algorithm 4 does not build any aggregates that cross over the borders of our disjoint partitioning. On the fine level, we rely on the user (or third party software) providing our solver with a reasonable partitioning of the global matrices and vectors onto the available processes. Often this partitioning will not take weak connection in the matrix graph into account. If we would continue the coarsening until no further decoupled aggregation is possible, the local part of the global matrix would either only have very few entries, entries mainly coupled by weak connections, e.g. for anisotropic problems, or both. This would cause a break down of the coarsening rate and/or very few computations between communication steps with small messages.

The currently available supercomputers, like the Blue Gene type systems of IBM, already make hundreds of thousands of cores available for usage. Even if the coarsest level system only has a few unknowns per core, the global system still has several hundred thousand unknowns. Solving such a coarse level system in parallel would mean doing very few floating point operations between many communication steps. Therefore this computation would be limited by the available bandwidth and latency of the communication network.

To overcome these problems, there is the option to agglomerate the data onto fewer processes. If agglomeration is not activated, all processes will compute on the coarsest level and a parallel Krylov method preconditioned with the smoother is used as a solver. Otherwise, whenever the average number of unknowns per core on a level drops below the prescribed coarsening target a new partitioning is computed using ParMETIS (cf. [23, 19]), a parallel graph partitioning software. We support two different choices for the input graph of the partitioning. The logically most reasonable is to use the weighted graph of the global matrix. Its edge weights are set to 1 for edges that are considered strong by our strength of connection measure and to 0 otherwise. This tells the graph partitioning software that weak connections can be cut at no cost and leads to partitionings that should keep small connected regions on one process. We believe that this approach results in sufficient coupling of strongly connected unknowns on coarser grids.

Unfortunately, at least as of version 3.1.1 ParMETIS uses a dense array of size $\#\mathcal{P} \times \#\mathcal{P}$ internally to capture all possible adjacencies between processes. This results in running out of memory on systems with very many cores (like the IBM Blue Gene/P). To prevent this we use the vertex-weighted graph of the communication pattern used in the parallel linear operator as input. Each process represents a vertex in the graph. The weight of the vertex is the number of matrix rows stored on this process. Edges appear only between pairs of vertices associated with processes that exchange data. This graph is gathered on one master process and the repartitioning is computed with the recursive graph repartitioning routine from sequential METIS. Then the data (matrices and vectors) of all processes associated with vertices in one partition, is agglomerated on one process and the others become idle on coarser levels. Obviously this is a sequential bottleneck of our method. It will improve once massively parallel

graph partitioning tool become available.

This kind of agglomeration is repeated on subsequent levels until there is only one participating process on the coarsest level. We can now use a sequential sparse direct solver as coarse level solver.

In Figure 1, the interplay of the coarsening and the data agglomeration process is sketched. Each node represents a stored matrix. Next to it the level index is written. As before the index 0 denotes the finest level. Note that on each level, where data agglomeration happens, some processes store two matrices, an agglomerated and a non-agglomerated one. The latter is marked with an inverted comma after the level number.

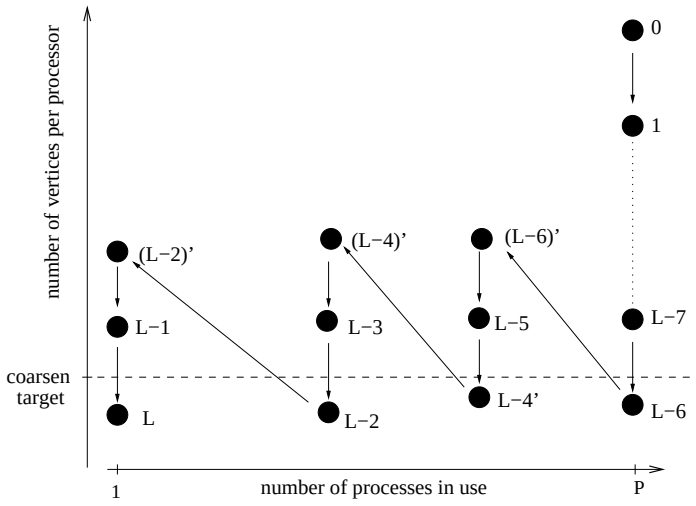

Figure 1: Data agglomeration

Whenever data agglomeration happened, the parallel smoothers use the not yet agglomerated matrix. The agglomerated matrix is only needed for the coarsening to the next level.

4 Implementational Details

The described algorithm is implemented in the "Distributed and Unified Numerics Environment" (DUNE) [7, 14, 5, 4]. As the components of this library are the main cause for the good performance of our method, we shortly introduce the two main building blocks of our AMG method: the parallel index sets, and the "Iterative Solver Template Library" (ISTL) [9, 10].

4.1 Parallel Index Sets

Our description of the parallelization in Section 3 is based on parallel finite index sets. This natural representation is directly built into our software and

used for the communication. We will only shortly sketch the relevant parts of the implementation. For a complete description of the parallel index set software see [11].

Each process p stores for each level one mapping of the corresponding index set $\widetilde{I}_{(p)}$ to $\{0,\dots,|\widetilde{I}_{(p)}|-1\}$. This mapping allows for using the efficient local matrix and vector data structures of ISTL to store the data and allows for direct random access. For every entry in $\widetilde{I}_{(p)}$ an additional marker is stored that lets us identify whether the index belongs to $\widetilde{I}_{(p)} \setminus I_{(p)}$ or to $I_{(p)}$. The mapping is represented by a custom container, that provides iterators over the entries. The key type used for this mapping is not limited to builtin integers but can be any integral numeric type. This allows us to realize keys with enough bits to represent integers bigger than $1.3 \cdot 10^{11}$.

Using these index sets all the necessary communication patterns are precomputed. The marker allows us for example to send data associated with $I_{(p)}$ for every process $p \in \mathcal{P}$ to all processes $q \in \mathcal{P}$ with $\widetilde{I}_{(q)} \cap I_{(p)} \neq \emptyset$. This kind of communication is used for the parallel application of the linear operator, the parallel smoother, and the communication of the aggregate information after the decoupled aggregation of one level. These communication patterns are implemented independent of the communicated type. The same pattern can be used to send for example vector entries of type double or the aggregate numbers represented by an arbitrarily sized integer type. During the communication step we collect all data for each such pair p, q of processes in a buffer and send all messages simultaneously using asynchronous communication of MPI. This keeps the number of messages as low as possible and at the same time uses the maximal message size possible for the problem. This reduces negative effects of network latency. As described already in Subsections 3.1 and 3.2 only one such communication step is necessary for each application of the linear operator or smoother. For the three dimensional model problems of the next section each process sends and receives one message to and from at most eight neighboring processes. The size of the mesage is smaller than 152 kByte.

It is even possible to use two different distributions $\bigcup_{p\in\mathcal{P}} \widetilde{J}_{(p)} = I$ and $\bigcup_{p\in\mathcal{Q}} \widetilde{I}_{(p)} = I$ as source and target of the communication. Whenever data agglomeration occurred on a level, we have two such distributions with $\#\mathcal{P} > \#\mathcal{Q}$. To collect data we send data associated to $I_{(p)}$ for every process $p \in \mathcal{P}$ to all processes $q \in \mathcal{Q}$ with $I_q \cap I_{(p)} \neq \emptyset$, when gathering data to fewer processes.

4.2 Efficient Local Linear Algebra

The "Iterative Solver Template Library" (ISTL) [9] is designed specifically for linear systems originating from the discretization of partial differential equations. An important application area are systems of PDEs. They often exhibit a natural block structure. The user of our method can either neglect this block structure like with most other libraries. The linear system is then simply resembled by a sparse matrix with scalar entries. In addition our method also supports a block-wise treatment of the unknowns, where all unknowns associ-

ated with the same discretization entity are grouped together. These groups must have the same size for all entities. The couplings between the grouped unknowns are represented by small dense matrices. The unknowns themselves in small vectors. The size of the matrices and vectors is known already at compile time.

ISTL offers specialized data structures for these and in addition supports block recursive sparse matrices of arbitrary recursion level. Using generic programming techniques the library lets the compiler optimize the code for the data structures during compilation. The available preconditioners and smoothers are implemented such that the same code supports arbitrary block recursion levels.

Therefore our method naturally supports so-called point-based AMG, where each matrix entry is a small dense matrix by itself. The graph used during the coarsening in Section 2.1 is the graph of the block matrix and the weight functions used in the criterions (3) and (4) are functions that turn the matrix blocks into scalars, such as the row-sum or Frobenius norm. The user only has to select the appropriate matrix and vector data structures and the smoother automatically becomes a block-smoother due to generic programming with templates.

5 Numerical Results

In this section we present scalability results for two model problems. First we solve simple Laplace and Poisson problems. Then we take a look at a heterogeneous model problem with highly variable coefficients. We perform our test on two different hardware platforms: a super-computer from IBM and a recent of-the-shelf Linux cluster.

The first machine is JUGENE located at the Forschungszentrum in Jülich, Germany. JUGENE is a Blue Gene/P machine manufactured by IBM that provides more than one petaflops as overall peak performance. Each compute node uses a 850 MHz PowerPC 450 quad-core CPU and provides 2GB of main memory with a bandwidth of 13.6 GB/s. The main interconnect is a 3D-Torus network for point to point message passing with a peak hardware bandwidth of 425MB/s in each direction of each torus link and a total of 5.1 GB/s of bidirectional bandwidth for each node. Additionally there are a global collective and a global barrier network. For comparison we also performed some tests on helics3a at Heidelberg University, an of-the-shelf Linux cluster consisting of 32 compute nodes with four AMD Opteron 6212 CPUs providing eight cores, each at 2.6 GHz. Each node utilizes 128 GB DDR3 RAM at 1333Mhz as main memory. The Infiniband network interconnect is a Mellanox 40G QDR single port PCIe Interconnect QSFP with 40 GB/s bidirectional bandwidths.

We start the analysis of our method by solving the Laplace equation, i.e. $K \equiv I$, with zero Dirichlet boundary conditions everywhere. The results of the weak scalability test can be found in Table 1. For the discretization we used a cell-centered finite volume scheme with 80^3 cells per participating core. Note that the biggest problem computed contains more than $1.34 \cdot 10^{11}$ unknowns.

The problems are discretized on a structured cube grid with uniform grid spacing h. We use one step of the V-cycle of the multigrid method as a preconditioner in a BiCGSTAB solver. For pre- and post-smoothing we apply one step of hybrid symmetric Gauss-Seidel. We measure the number of iterations (labelled It) to achieve a relative reduction of the Euclidian norm of the residual of 10^{-8}. Note that in each iteration the preconditioner is applied twice. We measure the number of grid levels (labelled lev.), the time needed per iteration (labelled TIt), the time for building the AMG hierarchy (labelled TB), the time needed for solving the linear system (labelled TS), and the total time needed to solution (labelled TT) including setup and solve phase depending on the number of processors (which is proportional to the number of grid cells $1/h$ cubed). Time is always measured in seconds.

procs	1/h	lev.	TB	TS	It	TIt	TT
1	80	5	19.86	31.91	8	3.989	51.77
8	160	6	27.7	46.4	10	4.64	74.2
64	320	7	74.1	49.3	10	4.93	123
512	640	8	76.91	60.2	12	5.017	137.1
4096	1280	10	81.31	64.45	13	4.958	145.8
32768	2560	11	92.75	65.55	13	5.042	158.3
262144	5120	12	188.5	67.66	13	5.205	256.2

Table 1: Laplace Problem 3D on JUGENE: Weak Scalability

Clearly, the time needed per iteration scales very well. When using nearly the whole machine in the run with 262,144 processes, we still reach an efficiency of about 77%. Due to the slight increase in the number of iterations the efficiency of the solution phase is about 47%. Unfortunately, the hierarchy building does not scale as well. This has different components, which can best be distinguished by an analysis of the time needed for some phases of the coarsening with agglomeration, which is displayed in Table 2.

procs	lev.	no.	TG	TM	TR
1	5	0	0.00	0.00	19.86
8	6	1	0.04	0.39	27.27
64	7	2	0.31	1.36	72.43
512	8	2	0.81	2.00	74.10
4096	10	3	2.18	3.04	75.46
32768	11	3	10.57	4.26	77.56
262144	12	4	98.23	4.65	85.71

Table 2: Time needed for Agglomeration and Coarsening(Laplace 3D)

In the table the column labelled "no." contains the number of data agglomeration steps, the column labelled "TG" contains the time spend in preparing the global graph on one process, partitioning it with METIS, and creating the communication infrastructure. The column "TM" contains the time needed for

redistributing the matrix data to the new partitions, and the column labelled "TR" contains the total time needed for the rest of the coarsening including the time for the factorization of the matrix on the coarsest level using SuperLU. If we directly agglomerate all data to one process, we do not use METIS as the repartitioning scheme is already known in advance. With an increasing number of processes the time spent for computing the graph repartitioning increases much faster than the time needed for the redistribution of the data. It turns out that this is one of the main bottlenecks, especially with very high processor numbers. However, there is also a markable increase in the time needed for the coarsening process itself. Without agglomeration the creation of the coarsest matrices would take less and less time as the total number of entries to be aggregated decreases. However, after each redistribution step the number of matrix entries per processor still participating in the computation step is increasing again. Thus the build time has a $log(P)$ dependency. The time TR needed for the coarsening increases much more whenever an additional level of agglomeration is needed.

procs	1/h	lev.	TB	TS	It	TIt	TT
1	190	5	37.97	71.77	8	8.97	109.7
8	380	6	50.90	211.39	14	15.10	262.0
64	760	8	60.00	243.23	15	16.20	303.0
512	1520	9	66.20	247.75	15	16.50	314.0

Table 3: Poisson Problem 3D on helics3a: Weak Scalability

We perform a slightly modified test on helics3a, where the Poisson problem is solved, described by

$$\begin{aligned} -\Delta u &= (6 - 4\|x\|)e^{\|x\|} && \text{in } \Omega = (0,1)^3, \\ u &= e^{\|x\|} && \text{on } \partial\Omega \end{aligned}$$

As helics3a has more main memory per core it is possible to use a grid which has more than eight times as many grid cells per core than on JUGENE.

The results of the weak scalability test can be found in Table 3. Please note that despite the larger problem per process the same number of levels in the matrix hierarchy as before is constructed. This is equivalent to an eight times larger problem on the coarsest level. The build time scales much better under these circumstances. This has two reasons. First, the size of the coarse grid problem after agglomeration is smaller compared to the large number of unknowns per processor. Secondly, the fraction of TR needed for the matrix decomposition is higher due to the larger matrix on the coarsest level, which reduces the influence of the graph partitioning and redistribution.

On helics3a with its much faster processor cores compared to JUGENE, the memory bandwidth becomes the limiting factor for the solution phase. While for the case of eight processes it would in principle be possible to distribute the processes in a way that each process still has full memory bandwidth, we did

not exploit this possibility as it is very tedious to achieve such a distribution and as it is no longer possible for the case of 64 or more processes anyhow. With the automatic process placement of the operating system processes will share a memory controller already in the case of 8 processes, which is reflected in the notable increase of the time per iteration. As expected the time per iteration is only slightly increasing when using even more processes. The hierarchy building is much less affected by the memory bandwidth limitation.

In addition we perform a strong scalability test on helics3a where the total problem size stays constant while the number of cores used increases. In this test we use decoupled coarsening until we reach the coarsening target and then agglomerate all the data onto one process at once and solve the coarse level system there. The results can be seen in Tables 4 and 5. Note, that when using 512 processes our method still has an efficiency of 27%. Again for the time needed for the solution phase (column TS) the biggest drop in efficiency occurs when using eight instead of one core due to the limited memory bandwidth. The setup phase (column TB) is not limited as much by it and scales much better than on JUGENE.

procs	TB	TS	It	TIt	TT
1	102.60	166.20	6	27.70	268.80
8	14.00	35.80	8	4.47	49.80
64	2.02	5.06	9	0.56	7.08
512	0.73	1.16	8	0.15	1.89

Table 4: Poisson Problem 3D on helics3a: Strong Scalability

procs	TB	TS	TIt	TT
8	0.92	0.58	0.77	0.68
64	0.79	0.51	0.77	0.59
512	0.27	0.28	0.37	0.28

Table 5: Poisson Problem 3D on helics3a: Strong Efficiency

The last model problem we investigate is the diffusion problem

$$\nabla \cdot (k(\mathbf{x})\nabla x) = f$$

on the unit cube $[0,1]^3$ with Dirichlet boundary conditions and jumps in the diffusion coefficient as proposed in [17]. Into the unit cube a smaller cube with width .8 is centered such that all faces are parallel to the faces of the enclosing cube. The diffusion coefficient k in this smaller cube is 10^3. Outside of the small cube $k = 1$ holds except for cubes with width 0.1 that are placed in all edges of the unit cube. There the diffusion coefficient is 10^{-2}. Again we use a cell-centered discretization scheme and the same settings as before for the AMG. The results of a weak scalability test on 64 racks of JUGENE can be found in Table 6.

procs	1/h	lev.	TB	TS	It	TIt	TT
1	80	5	19.88	36.27	9	4.029	56.15
8	160	6	27.8	48.9	10	4.89	76.7
64	320	7	74.4	59.6	12	4.96	134
512	640	8	78.04	72.67	14	5.191	150.7
4096	1280	10	89.72	73.37	14	5.241	163.1
32768	2560	11	94.48	104.2	20	5.21	198.7
262144	5120	12	186.2	85.87	16	5.367	272.1

Table 6: Heterogeneous Diffusion Problem 3D on JUGENE: Weak Scalability

Compared to the Poisson problem the number of iterations increases more steeply due to the jumps in the diffusion coefficient. Again this is not due to the parallelization but due to the nature of the problem. The time needed for building the matrix hierarchy as well as the time needed for one iteration scale as for the previous problems. Compared to the AMG method used in [17] on an (now outdated) Blue Gene/L our method scales much better when used on Blue Gene/P. In small parts this might due to the new architecture and the bigger problem size per core used. But this cannot explain all the difference in the scaling behavior. Additionally, the large problem size is only possible because of the smaller memory foot-print of our method.

6 Summary and Conclusion

We have presented a parallel algebraic multigrid algorithm based on non-smoothed aggregation. During the setup phase it uses an elaborate heuristic aggregation algorithm to account for highly variable coefficients that appear in many application areas. Due to its simple piecewise constant interpolation between the levels, the memory consumption of the method is rather low and allows for solving problems with more than 10^{11} unknowns using 64 racks of an IBM Blue Gene/P. The parallelization of the solution phase scales well for up to nearly 300,000 cores. Although there is a sequential bottleneck in the setup phase of the method due to the lack of scalable parallel graph partitioning software, the method still scales very well in terms of total time to solution. For comparison see [2] where during a weak scalability test for the Laplace problem on an IBM Blue Gene/P the total solution time for interpolation AMG increases by more than a factor 2 when going from 128 to 128,000 processes. In contrast, for our method with the same increase in total solution time we can go from 64 to up to 262,144 processes during weak scaling.

We also have shown that our solver scales reasonably well even for hard problems that have highly variable coefficients. Even for modern clusters consisting out of multicore machines the method scales very well and is only limited by the available memory bandwidth per core.

Once scalable parallel graph partition software is available, the bottleneck of the sequential graph partitioning will disappear rendering the method even

more scalable.

References

[1] D. M. Alber and L. N. Olson, *Parallel coarse-grid selection*, Numer. Linear Algebra Appl., 14 (2007), pp. 611–643.

[2] A. H. Baker, R. D. Falgout, T. Gamblin, T. V. Kolev, M. Schulz, and U. M. Yang, *Scaling algebraic multigrid solvers: On the road to exascale*, in Competence in High Performance Computing 2010, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, eds., Springer Berlin Heidelberg, 2012, pp. 215–226.

[3] R. Bank, S. Lu, C. Tong, and P. Vassilevski, *Scalable parallel algebraic multigrid solvers*, Tech. Report UCRL-TR-210788, Lawrence Livermore National Laboratory, 2005.

[4] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. part II: implementation and test in DUNE*, Computing, 82 (2008), pp. 121–138.

[5] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. part I: abstract framework*, Computing, 82 (2008), pp. 103–119.

[6] P. Bastian, M. Blatt, and R. Scheichl, *Algebraic multigrid for discontinuous galerkin discretizations of heterogeneous elliptic problems*, Numer. Linear Algebra Appl., 19 (2012), pp. 367–388.

[7] P. Bastian, M. Droske, C. Engwer, R. Klöfkorn, T. Neubauer, M. Ohlberger, and M. Rumpf, *Towards a unified framework for scientific computing*, in Domain Decomposition Methods in Science and Engineering, R. Kornhuber, R. Hoppe, J. Priaux, O. W. O. Pironneau, and J. Xu, eds., vol. 40 of LNCSE, Springer-Verlag, 2005, pp. 167–174.

[8] M. Blatt, *A Parallel Algebraic Multigrid Method for Elliptic Problems with Highly Discontinuous Coefficients*, PhD thesis, Ruprecht–Karls–Universität Heidelberg, 2010.

[9] M. Blatt and P. Bastian, *The iterative solver template library*, in Applied Parallel Computing. State of the Art in Scientific Computing, B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, eds., vol. 4699 of Lecture Notes in Computer Science, Springer, 2007, pp. 666–675.

[10] ——, *On the generic parallelisation of iterative solvers for the finite element method*, Int. J. Comput. Sci. Engrg., 4 (2008), pp. 56–69.

[11] ——, *C++ components describing parallel domain decomposition and communication*, Int. J. Parallel Emergent Distrib. Syst., 24 (2009), pp. 467–477.

[12] D. Braess, *Towards algebraic multigrid for elliptic problems of second order*, Computing, 55 (1995), pp. 379–393.

[13] H. De Sterck, U. Meier-Yang, and J. Heys, *Reducing complexity in parallel algebraic multigrid preconditioners*, SIAM. J. Matrix Anal. and Appl., 27 (2006), pp. 1019–1039.

[14] DUNE. `http://www.dune-project.org/`.

[15] R. Falgout, V. Henson, J. Jones, and U. Meier-Yang, *Boomer AMG: A parallel implementation of algebraic multigrid*, Tech. Report UCRL-MI-133583, Lawrence Livermore National Laboratory, 1999.

[16] P. O. Frederickson and O. A. McByran, *Parallel superconvergent multigrid*, tech. report, Cornell University, 1987.

[17] M. Griebel, B. Metsch, and M. A. Schweitzer, *Coarse grid classification: AMG on parallel computers*, in NIC Symposium 2008, G. Münster, D. Wolf, and M. Kremer, eds., vol. 39 of NIC Series, February 2008, pp. 299–306.

[18] W. Joubert and J. Cullum, *Scalable algebraic multigrid on 3500 processors*, Electronic Transactions on Numerical Analysis, 23 (2006), pp. 105–128.

[19] G. Karypis and V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, J. Parallel Distrib. Comput., 48 (1998), pp. 71–95.

[20] A. Krechel and K. Stüben, *Parallel algebraic multigrid based on subdomain blocking*, Parallel Comput., 27 (2001), pp. 1009 – 1031.

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

[22] Y. Notay, *An aggregation-based algebraic multigrid method*, Electron. Trans. Numer. Anal., 37 (2010), pp. 123–146.

[23] ParMETIS. Available online at `http://www-users.cs.umn.edu/karypis/metis/`.

[24] M. Raw, *A coupled algebraic multigrid method for the 3d Navier-Stokes equations*, in Fast Solvers for Flow Problems, Proceedings of the 10th GAMM-Seminar, vol. 49 of Notes on Numerical Fluid Mechanics, Vieweg-Verlag, Braunschweig, Wiesbaden, 1985.

[25] M. Raw, *Robustness of coupled algebraic multigrid for the Navier-Stokes equations*, AIAA Paper no 960297, (1996).

[26] J. Ruge and K. Stüben, *Algebraic multigrid*, in Multigrid Methods, S. F. McCormick, ed., SIAM Philadelphia, 1987, ch. 4, pp. 73–130.

[27] R. Scheichl and E. Vainikko, *Additive Schwarz with aggregation-based coarsening for elliptic problems with highly variable coefficients*, Computing, 80 (2007), pp. 319–343.

[28] K. Stüben, *Algebraic multigrid (amg): An introduction with applications*, Computing, (1999), pp. 1–127.

[29] R. Tuminaro and C. Tong, *Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines*, in in SuperComputing 2000 Proceedings, 2000.

[30] P. Vaněk, J. Mandel, and M. Brezina, *Algebraic multi–grid by smoothed aggregation for second and forth order elliptic problems*, Computing, 56 (1996), pp. 179–196.

[31] P. Vaněk, J. Mandel, and M. Brezina, *Algebraic multigrid based on smoothed aggregation for second and fourth order problems*, Computing, 56 (1996), pp. 179–196.