Parallel AMG on Distributed Memory Computers [1]

Gundolf Haase [a] Michael Kuhn [b] Stefan Reitzinger [b]

[a] *Johannes Kepler University Linz,*
Institute for Analysis and Computational Mathematics,
Altenberger Str. 69, A–4040 Linz, Austria

[b] *Johannes Kepler University Linz,*
SFB F013 "Numerical and Symbolic Scientific Computing",
Freistädter Str. 313, A–4040 Linz, Austria

Abstract

Algebraic Multigrid (AMG) methods are well suited as preconditioners for iterative solvers of linear systems of equations which are sparse, symmetric positive definite and stem from a finite element (FE) discretization of a 2^{nd} order elliptic partial differential equation (PDE) or a system of PDEs. Since preconditioners based on AMG are very efficient, additional speedup can only be achieved by parallelization. In this paper we propose a general parallel AMG algorithm which is well suited for distributed memory computers. The algorithm is based on domain decomposition ideas and allows overlapping and non-overlapping data decompositions. This paper pays special attention to the coarsening strategy which has to be adapted in the parallel case. Moreover, a general framework of data distribution gives rise to a construction scheme for the prolongation operators.

Results of numerical studies on parallel machines with distributed memory are presented which show the high efficiency of the approach.

Keywords : Algebraic Multigrid, Finite Element Method, Domain Decomposition Data Distribution, Parallel Iterative Solvers, Parallel Multigrid Preconditioners, Distributed Memory.

---

[1] This research has been supported by the Austrian Science Fund FWF within the SFB "Numerical and Symbolic Scientific Computing" under the grant SFB F013

1 Introduction

The fast and efficient solution of linear systems of equations is a key task in the solution process in many practical applications arising in science and engineering. Such systems arise after an appropriate discretization of the underlying partial differential equation (PDE) or a system of PDEs. In the case of nonlinear or optimization problems linear systems have to be solved repeatedly as part of an outer iteration loop, e.g., a Newton iteration. If the size of such systems, i.e., the number of unknowns, grows, it is important to use algorithms of optimal complexity. Both, memory and time consumption should be proportional to the number of unknowns. Krylov subspace methods together with AMG-based preconditioners fulfill these requirements. AMG methods are of special interest if the more classical geometric multigrid cannot be applied. Two reasons for using AMG are:

(1) The discretization provides no hierarchy of meshes.
(2) The coarsest grid of a geometric multigrid method is too large to be solved efficiently by a direct or classical iterative solver.

In contrast to geometric multigrid methods where a grid hierarchy is required explicitly, AMG constructs the matrix hierarchy and prolongation operators just by knowing the stiffness matrix. In this way, systems with up to several millions of unknowns can be handled even on single processor machines, nowadays. If additional speedup is required, the use of parallel computers is the only way to overcome the limitations. Hereby, distributed memory computers, e.g. PC clusters, are of special practical interest because of the availability and low costs of such machines.

The parallelization based on domain decomposition ideas has been proved to be very efficient. Domain decomposition methods and parallel geometric multigrid methods, both requiring a hierarchy of meshes, have been investigated in [9,12]. In this paper we apply similar ideas for the design of a general parallel AMG method. There are several sequential version of AMG methods like [1,13,22,24]. The methods differ mainly in the setup phase, i.e., construction of the matrix hierarchy and prolongation operators. The multilevel cycle is then realized in the classical way and performs well if all components are properly chosen. Parallel versions of AMG are rather rare. Some of them are [3,4,14]. In contrast to the parallelization approaches in [3], we base our parallelization strategy for AMG on the non-overlapping domain decomposition. Special attention is paid to a modular implementation of our parallel code, such that other AMG codes fit in the parallelization concept as well.

By now we concentrate on the efficient solution of

$$K_h \underline{u}_h = \underline{f}_h \tag{1}$$

where $K_h \in \mathbb{R}^{N_h \times N_h}$ denotes the stiffness matrix, $\underline{u}_h \in \mathbb{R}^{N_h}$ the solution vector of nodal unknowns and $\underline{f}_h \in \mathbb{R}^{N_h}$ the load vector. K_h is assumed to be symmetric positive definite (SPD) and has sparse matrix structure. The number of unknowns is denoted by N_h and it is related to the discretization parameter h (mesh-width) by $N_h = \mathcal{O}(h^{-d})$ in the case of a uniform discretization, where d is the spatial dimension. Hence N_h becomes very large if h is decreased in order to reduce the discretization error. We restrict ourselves to the problem class where the stiffness matrix K_h arises from a second order PDE (or a system of PDEs) with homogeneous Dirichlet boundary conditions in a bounded domain $\Omega \subset \mathbb{R}^d$ $d = 2, 3$ with sufficiently smooth boundary $\partial\Omega$. The variational form is formulated in an appropriate Sobolev-space $\mathbb{V}$ and is given by

$$\text{Find } u \in \mathbb{V}: \quad a(u,v) = \langle F, v\rangle \quad \forall v \in \mathbb{V} \tag{2}$$

with a symmetric, coercive bilinear form $a(u,v) : \mathbb{V} \times \mathbb{V} \mapsto \mathbb{R}$ and the duality product $\langle F, v\rangle : \mathbb{V}^* \times \mathbb{V} \mapsto \mathbb{R}$. A discretization of the domain Ω results in a finite element (FE) mesh τ_h and the index set of nodes ω_h. The FE-basis $\Phi = \left\{\varphi^{[j]}(x)\,,\, j \in \omega_h\right\}$ has local support spanning an FE-space $\mathbb{V}_h = \text{span}\Phi$ and changes (2) into

$$\text{Find } u_h \in \mathbb{V}_h \subseteq \mathbb{V}: \quad a(u_h, v_h) = \langle F, v_h\rangle \quad \forall v_h \in \mathbb{V}_h \ . \tag{3}$$

In this paper we always assume Lagrange FE-functions, i.e., $\mathbb{V} = (H_0^1(\Omega))^p$ $p = 1, 2, \ldots$. The FE-isomorphism guarantees the equivalence of (3) and (1), i.e., $\mathbb{V}_h \cong V_h := \mathbb{R}^{N_h}$.

The remaining paper is organized as follows. Section 2 introduces the sequential version of AMG and Section 3 provides some definitions and data types for the parallelization strategy from [5,10,12,17,18] now for non-overlapping and overlapping domain decompositions. Section 4 combines the previous two sections to derive a parallel AMG algorithm followed by numerical tests in Section 5. Finally, conclusions are drawn and further remarks are given.

2 Algebraic multigrid

In this section, we introduce the sequential version of AMG which is the starting point for the parallel algorithm. For the sake of simplicity we explain this method for the scalar case. The case of block systems will be handled similarly and can be found in [11].

As in geometric multigrid, the basic idea in AMG is to reduce high and low frequency components of the error by smoothing and coarse grid correction, respectively. In AMG, both, the matrix hierarchy and the prolongation operators are constructed just from the stiffness matrix K_h. In analogy, we will speak of "coarse grids" although these are purely virtual and do not have to supplied explicitly as coarse FE-meshes. Each AMG algorithm consists of the following components:

(1) Coarsening, i.e., define the splitting $\omega_h = \omega_C \cup \omega_F$ of ω_h into coarse and fine grid nodes ω_C and ω_F, respectively.
(2) Transfer operators: prolongation $P : V_H \mapsto V_h$ and restriction $R := P^T$.
(3) Definition of the coarse matrix $K_H := RK_hP$.
(4) Appropriate smoother for the considered problem class.

The most important issue to be discussed is the setup phase, i.e., the construction of the matrix hierarchy and the prolongation operators. We will give the explanation for a two grid method where h is related to the fine grid ($\ell = 1$) and H to the coarse grid ($\ell = 2$). A recursive application yields immediately a multilevel cycle.

In our case, the stiffness matrix K_h can be associated with an FE-grid, i.e., the diagonal entry of the i^{th} row of the matrix K_h is related to a grid point in ω_h and an off-diagonal entry is related to an edge in an FE-grid (see Fig. 1). First we investigate the coarsening process which has the task to reduce the

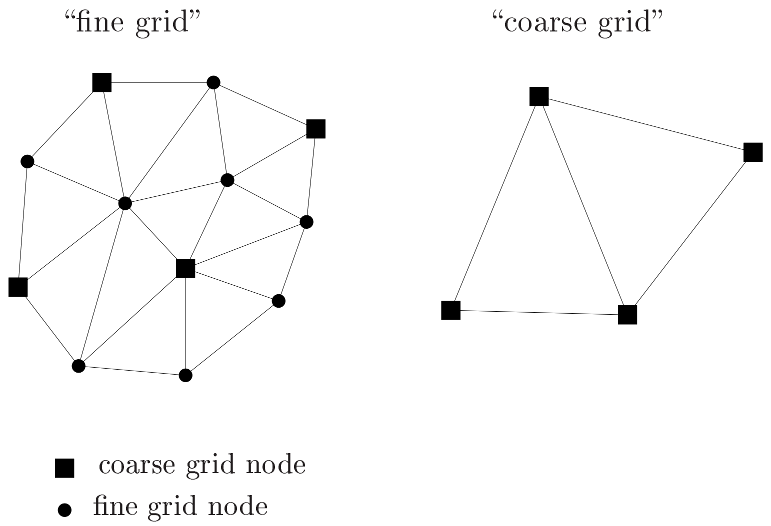

Figure 1. Illustration of a two grid method.

nodes such that $|\omega_C| < |\omega_h|$. Here, $|\omega|$ denotes the number of elements in the set ω. Motivated from Fig. 1, the grid points ω_h can be split into two disjoint subsets ω_C (coarse grid nodes) and ω_F (fine grid nodes), i.e.,

$$\omega_h = \omega_C \cup \omega_F \tag{4}$$

such that there are (almost) no direct connections between any two coarse grid nodes and the resulting number of coarse grid nodes is as large as possible. Instead of considering all connections between nodes being of the same rank, we introduce the following sets

$$N^i = \{j \mid k_{ij} \neq 0,\ i \neq j\} \tag{5}$$

$$S^i = \left\{j \in N^i \mid k_{ij} > coarse(K_h)\right\} \tag{6}$$

which are related to the neighbourhood and the strong connections around $i \in \omega_h$, respectively (see [22]). Here $coarse(K_h)$ is an appropriate cut-off (coarsening) function. The routine

$$\{S^i\}_{i=1}^{N_h} \leftarrow \textsc{GetStrong}(K_h)\,,$$

provides the sets $\{S^i\}$ for all $i \in \omega_h$. A splitting can be performed with the sets $\{S^i\}$ where $\{S^i\}$ is the set of all nodes, which can be agglomerated to a coarse grid node (for an algorithm see [13,22,24]). The corresponding routine is

$$(\omega_C, \omega_F) \leftarrow \textsc{Coarse}(\{S^i\}_{i=1}^{N_h}, \omega_h)\,.$$

Next the prolongation operator has to be defined correctly. We require that the prolongation operator $P : V_H \mapsto V_h$ has full rank. There are a lot of possibilities to define such transfer operators with pure algebraic information. For the construction we refer to [1,13,22]. The routine is denoted by

$$P \leftarrow \textsc{Weights}(\{S^i\}_{i=1}^{N_h}, K_h, \omega_C, \omega_F)\,.$$

The coarse grid matrix K_H is defined by the classical Galerkin method, i.e.,

$$K_H = P^T K_h P. \tag{7}$$

The corresponding routine is denoted by

$$K_H \leftarrow \textsc{Galerkin}(K_h, P)\,.$$

Hence, we can formulate the setup algorithm (Alg. 1) which is is a realization of the components (1)–(3) of our AMG method, see page 4. In Alg. 1 the parameter CoarseGrid is an appropriately chosen coarse-grid size for which a direct solver is applicable. The variable CoarseLevel stores the number of levels generated by the coarsening process until the size of the system is smaller than CoarseGrid. Initially, the setup function Setup(K_h, ω, ℓ) is called for $\ell = 1$ with the system matrix K_h and the set of all nodes ω_h.

Algorithm 1 Sequential Setup $\textsc{Setup}(K_h, \omega, \ell)$

if $|\omega| > \textsc{CoarseGrid}$ **then**
  $\mathsf{K}^\ell \leftarrow K_h$, $\omega^\ell \leftarrow \omega$
  $\{S^i\}^\ell \leftarrow \textsc{GetStrong}(K^\ell)$
  $(\omega_C, \omega_F) \leftarrow \textsc{Coarse}(\{S^i\}^\ell, \omega^\ell)$
  $P^\ell \leftarrow \textsc{Weights}(\{S^i\}^\ell, K^\ell, \omega_C, \omega_F)$, **free**$(\{S^i\}^\ell)$
  $\mathsf{K}^{\ell+1} \leftarrow 0$
  $K^{\ell+1} \leftarrow \textsc{Galerkin}(K^\ell, P^\ell)$
  $\omega^{\ell+1} \leftarrow \omega_C$, $\ell \leftarrow \ell + 1$
  $\textsc{Setup}(K^{\ell+1}, \omega^{\ell+1}, \ell + 1)$
else
  $\textsc{CoarseLevel} \leftarrow \ell$
end if

3 A unified data distribution concept

3.1 Notations

We use a non-overlapping domain decomposition for our approach, i.e., we decompose $\overline{\Omega}$ into P subdomains $\overline{\Omega}_s$ such that

$$\overline{\Omega} = \bigcup_{s=1}^{P} \overline{\Omega}_s \text{ with } \Omega_s \cap \Omega_q = \emptyset,\ \forall q \neq s,\ s, q = \overline{1, P}$$

holds. Each subdomain Ω_s is discretized by a mesh $\tau_{h,s}$, such that the whole triangulation $\tau_h = \bigcup\limits_{s=1}^{P} \tau_{h,s}$ of Ω is conform. A global finite element space $\mathbb{V}_h$ is defined with respect to τ_h and the local spaces $\mathbb{V}_{h,s}$ are projections of $\mathbb{V}_h$ onto $\tau_{h,s}$. The index set of nodes in Ω_s is denoted by ω_s. Note, that $\omega_h = \bigcup_{s=1}^{P} \omega_s$ but $N_h = |\omega_h| \leq \sum_{s=1}^{P} |\omega_s| =: \sum_{s=1}^{P} N_s$ holds, i.e., different subdomains may share unknowns although elements from different subdomains do not overlap.

The mapping of a vector $\underline{u}_h \in \mathbb{R}^{N_h}$ in global numbering onto a local vector $\underline{u}_s \in \mathbb{R}^{N_s}$ in subdomain $\overline{\Omega}_s$ $(s = \overline{1, P})$ is represented symbolically by subdomain connectivity matrices $\mathcal{A}_s : \mathbb{R}^{N_h} \mapsto \mathbb{R}^{N_s}$ with entries

$$\mathcal{A}_s^{[i,j]} := \begin{cases} 1 & \text{if} \quad j = \text{global number of } i \\ 0 & \text{else} \end{cases} \qquad \forall i \in \omega_s\,,\ \forall j \in \omega_h\ . \tag{8}$$

The transpose $\mathcal{A}_s^T$ of these binary matrices $\mathcal{A}_s$ maps a local vector back to the global one. The index set of all those subdomains, an unknown $u^{[j]}$, $j \in \omega_h$ belongs to, is denoted by

$$\sigma^{[j]} := \{s \mid u_h^{[j]} \in \overline{\Omega}_s\} = \{s \mid \exists i \in \omega_s : \mathcal{A}_s^{[i,j]} \neq 0\}\ . \tag{9}$$

We store the data of a vector component $u_h^{[i]}$ in the subdomain Ω_s if $s \in \sigma^{[i]}$. There are (at least) two opportunities to store those components and consequently that vector (see also Fig. 3.2.

- A vector $\underline{\mathfrak{u}}$ is called an accumulated vector if each vector component $\mathfrak{u}^{[i]}$ is stored in all subdomains Ω_s, $s \in \sigma^{[i]}$ with its full value. The local vectors $\underline{\mathfrak{u}}_s$ can be represented as

$$\underline{\mathfrak{u}}_s \;:=\; \mathcal{A}_s \cdot \underline{\mathfrak{u}} \;. \tag{10}$$

- A vector $\underline{\mathfrak{r}}$ is called a distributed vector if it is decomposed into local vectors $\underline{\mathfrak{r}}_s$ such that

$$\underline{\mathfrak{r}} \;=\; \sum_{s=1}^{P} \mathcal{A}_s^T \cdot \underline{\mathfrak{r}}_s \tag{11}$$

  holds, i.e., all subdomains Ω_s, $s \in \sigma^{[i]}$ store only $\underline{\mathfrak{r}}_s$ and possess a portion of the full vector value $\mathsf{r}^{[i]}$ which can be determined only by taking the sum in (11).

The conversion of a distributed vector $\underline{\mathfrak{v}}$ into an accumulated vector $\underline{\mathfrak{w}}$ can be done by evaluating the sum in (11) followed by the restriction (10), i.e.,

$$\underline{\mathfrak{w}} \leftarrow \underline{\mathfrak{v}} \qquad : \qquad \underline{\mathfrak{w}}_s := \mathcal{A}_s \cdot \underline{\mathfrak{w}} \;=\; \mathcal{A}_s \cdot \sum_{s=1}^{P} \mathcal{A}_s^T \cdot \underline{\mathfrak{v}}_s \;. \tag{12}$$

The conversion in the other direction is not unique - we prefer an equal weighted distribution of the accumulated vector. A matrix weighted distribution is also feasible in case of varying coefficients. The weights are chosen such that the re-conversion (12) will result in the original vector :

$$\underline{\mathfrak{v}} \leftarrow \underline{\mathfrak{w}} \qquad : \qquad \underline{\mathfrak{v}}_s := (R_s)^{-1} \cdot \underline{\mathfrak{w}}_s \;, \tag{13}$$

with $R_s = \operatorname*{diag}_{i\in\omega_s}\{|\sigma^{[i]}|\}$, i.e., $R_s^{[i,i]}$ stores the number of subdomains the component i belongs to.

The matrix defined in (1) can also be stored in two ways. With respect to an element-wise domain decomposition, we can store the FE-matrix accumulated or distributed.

- A matrix $\mathfrak{M}$ is called accumulated if its local restrictions $\mathfrak{M}_s$ possess the full entries of it, and we can write

$$\mathfrak{M}_s \;:=\; \mathcal{A}_s \cdot \mathfrak{M} \cdot \mathcal{A}_s^T \;. \tag{14}$$

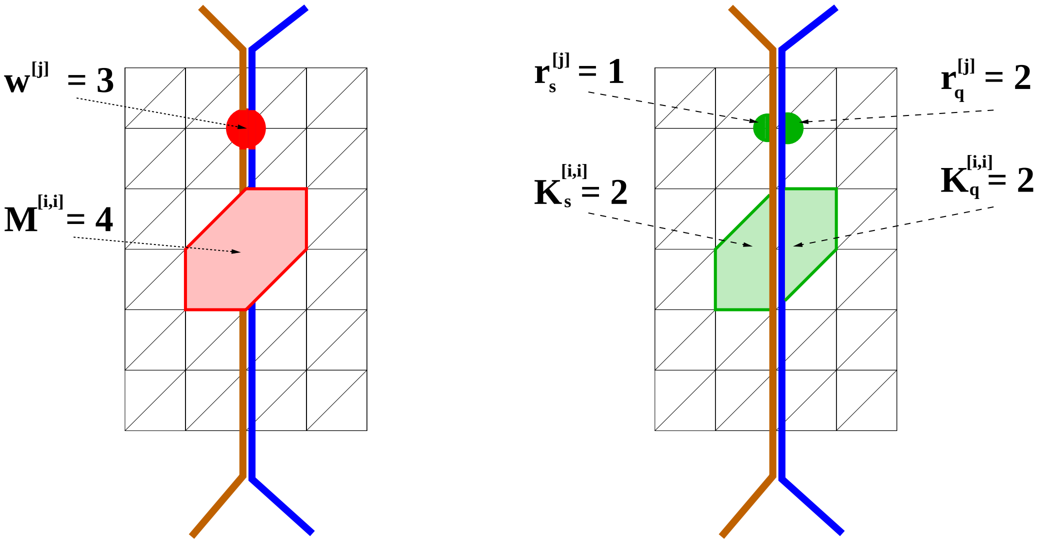

Figure 2. Illustration for accumulated and distributed vectors and matrices.

- We call a matrix K_h distributed if we have locally stored matrices K_s such that

$$\mathsf{K}_h \;:=\; \sum_{s=1}^{P} \mathcal{A}_s^T \cdot \mathsf{K}_s \cdot \mathcal{A}_s \tag{15}$$

  holds, i.e., each subdomain $\overline{\Omega}_s$ stores only a part of its full entries.

We obtain distributed matrices K_s automatically after the local FE-accumulation with respect to $\mathbb{V}_{h,s}$, due to our non-overlapping construction principle. The sum in (15) is equivalent to a global matrix accumulation which will not be carried out in our approach.

3.3 Basic operations

This section collects some result from previous papers, for details see [5–7,10]. It is trivial that additive combinations of vectors from the same type do not require any communication. The inner product of different type vectors requires one global reduce operation of the local inner products:

$$\langle \underline{\mathfrak{w}}, \underline{\mathfrak{r}} \rangle \;=\; \underline{\mathfrak{w}}^T \underline{\mathfrak{r}} \;=\; \underline{\mathfrak{w}}^T \sum_{s=1}^{P} \mathcal{A}_s^T \underline{\mathfrak{r}}_s \;=\; \sum_{s=1}^{P} (\mathcal{A}_s \underline{\mathfrak{w}})^T \underline{\mathfrak{r}}_s \;=\; \sum_{s=1}^{P} \langle \underline{\mathfrak{w}}_s, \underline{\mathfrak{r}}_s \rangle \quad . \tag{16}$$

Any other combination of vector types requires more effort. The multiplication of a distributed matrix with an accumulated vector

$$\mathsf{K}_h \cdot \underline{\mathfrak{w}} \;=\; \sum_{s=1}^{P} \mathcal{A}_s^T \mathsf{K}_s \mathcal{A}_s \cdot \underline{\mathfrak{w}} \;=\; \sum_{s=1}^{P} \mathcal{A}_s^T \left(\mathsf{K}_s \cdot \underline{\mathfrak{w}}_s\right) \;=\; \sum_{s=1}^{P} \mathcal{A}_s^T \underline{\mathsf{v}}_s \;=\; \underline{\mathsf{v}} \;, \tag{17}$$

results in a distributed vector. The realization requires no communication at all because we only have to compute locally $\underline{\mathsf{v}}_s = \mathsf{K}_s \cdot \underline{\mathfrak{w}}_s$.

The situation changes if we use an accumulated matrix $\mathfrak{M}$. Here, we have to ensure that the pattern of $\mathfrak{M}$ fulfills the condition

$$\forall i,j \in \omega_h : \qquad \sigma^{[i]} \not\subseteq \sigma^{[j]} \implies \mathfrak{M}^{[i,j]} = 0 \ . \tag{18}$$

If the pattern condition (18) holds then the operations

$$\underline{\mathfrak{w}} \ = \ \mathfrak{M} \cdot \underline{\mathfrak{u}} \qquad \text{and} \qquad \underline{\mathsf{d}} \ = \ \mathfrak{M}^T \cdot \underline{\mathsf{r}} \tag{19}$$

can be performed locally without any communication, i.e., $\forall s = \overline{1,P}$

$$\underline{\mathfrak{w}}_s \ = \ \mathfrak{M}_s \cdot \underline{\mathfrak{u}}_s \qquad \text{and} \qquad \underline{\mathsf{d}}_s \ = \ \mathfrak{M}_s^T \cdot \underline{\mathsf{r}}_s \ . \tag{20}$$

Under the same assumption, a special product of three matrices can be performed locally without communication, see also [7],

$$\mathsf{K}_H := \mathfrak{M}^T \cdot \mathsf{K}_h \cdot \mathfrak{M} \qquad \text{, i.e.,} \qquad \mathsf{K}_{H,s} := \mathfrak{M}_s^T \cdot \mathsf{K}_s \cdot \mathfrak{M}_s \ . \tag{21}$$

3.4 Basic algorithms

The operations (16) and (17) allow already to formulate a parallel preconditioned conjugate gradient (PCG) algorithm for solving the matrix equation $K_h \underline{u}_h = \underline{f}_h$ with a preconditioner C_K.

Algorithm 2 PCG algorithm PCG($\mathsf{K}_h, \underline{\mathfrak{u}}, \underline{\mathsf{f}}, C_K$)

repeat
- $\underline{\mathsf{v}} \leftarrow \mathsf{K}_h \cdot \underline{\mathfrak{s}}$
- $\alpha \leftarrow \sigma / \langle \underline{\mathfrak{s}}, \underline{\mathsf{v}} \rangle$
- $\underline{\mathfrak{u}} \leftarrow \underline{\mathfrak{u}} + \alpha \cdot \underline{\mathfrak{s}}$
- $\underline{\mathsf{r}} \leftarrow \underline{\mathsf{r}} - \alpha \cdot \underline{\mathsf{v}}$
- $\underline{\mathfrak{w}} \Leftarrow C_K^{-1} \cdot \underline{\mathsf{r}}$
- $\sigma \leftarrow \langle \underline{\mathfrak{w}}, \underline{\mathsf{r}} \rangle$
- $\beta \leftarrow \sigma / \sigma_{\text{old}} \quad , \quad \sigma_{\text{old}} \leftarrow \sigma$
- $\underline{\mathfrak{s}} \leftarrow \underline{\mathfrak{w}} + \beta \cdot \underline{\mathfrak{s}}$

until termination

Besides the inner products, only the preconditioning step $\underline{\mathfrak{w}} \Leftarrow C_K^{-1} \cdot \underline{\mathsf{r}}$ involves communication indicated by using $\Leftarrow$ instead of $\leftarrow$. In the case of $C_K = I$, i.e., no preconditioning, this step reduces to a type conversion (12) involving communication. We require that the communication costs for applying any other preconditioner C_K^{-1} are in the same range.

One possible choice for the preconditioner is $C_K^{-1} = (I - M_K)K_h^{-1}$, with M_K being the multigrid iteration operator for K_h. The parallel multigrid iteration is presented in Alg. 3, where ℓ denotes the level such that $\ell = 1$ stands for the finest grid. The algorithm applies the parallel version of the smoother $\text{SMOOTH}(\mathsf{K}_h, \underline{\mathfrak{u}}, \underline{\mathsf{f}})$ and its transpose, e.g., a block Jacobi smoother with Gauss-Seidel smoothing in blocks containing interior unknowns of the subdomains. Furthermore, the interpolation $\mathfrak{P}$ has to fulfill the pattern condition (18) and we take $\mathfrak{P}^T$ as restriction. The coarse grid system can be solved directly (either in parallel or sequentially on one processor) or again by some iterative method similar to the PCG in Alg. 2.

Algorithm 3 Parallel multigrid $\text{PMG}(\mathsf{K}_h, \underline{\mathfrak{u}}, \underline{\mathsf{f}}, \ell)$

if $\ell == \text{COARSELEVEL}$ **then**
  $\underline{\mathfrak{u}} \Leftarrow \text{SOLVE}\left(\sum_{s=1}^{P} \mathcal{A}_s^T \mathsf{K}_s \mathcal{A}_s \cdot \underline{\mathfrak{u}} \; = \; \underline{\mathsf{f}} \right)$
else
  $\underline{\widetilde{\mathfrak{u}}} \Leftarrow \text{SMOOTH}(\mathsf{K}_h, \underline{\mathfrak{u}}, \underline{\mathsf{f}})$
  $\underline{\mathsf{d}} \leftarrow \underline{\mathsf{f}} - \mathsf{K}_h \cdot \underline{\widetilde{\mathfrak{u}}}$
  $\underline{\mathsf{d}}^H \leftarrow \mathfrak{P}^T \cdot \underline{\mathsf{d}}$
  $\underline{\mathfrak{w}}^H \leftarrow 0$
  $\underline{\mathfrak{w}}^H \Leftarrow \text{PMG}(\mathsf{K}_H, \underline{\mathfrak{w}}^H, \underline{\mathsf{d}}^H, \ell + 1)$
  $\underline{\mathfrak{w}} \leftarrow \mathfrak{P} \cdot \underline{\mathfrak{w}}^H$
  $\underline{\widehat{\mathfrak{u}}} \leftarrow \underline{\widetilde{\mathfrak{u}}} + \underline{\mathfrak{w}}$
  $\underline{\mathfrak{u}} \Leftarrow \text{SMOOTH}^T(\mathsf{K}_h, \underline{\widehat{\mathfrak{u}}}, \underline{\mathsf{f}})$
end if

Despite the coarse grid solver, only the smoothing sweeps require communication.

4 The parallel AMG algorithm and its implementation

It follows from Alg. 3, the Galerkin approach (7) and rule (21) that the crucial point for a recursive parallel AMG algorithm consists in guaranteeing the pattern condition (18) for the interpolation matrix $\mathfrak{P}$. This pattern is controlled by the strong connections $\{S^i\}$ as input parameters in WEIGHTS on page 5. Therefore, we have to control the sets of strong connections in the coarsening step to ensure the required interpolation pattern, see also [7].

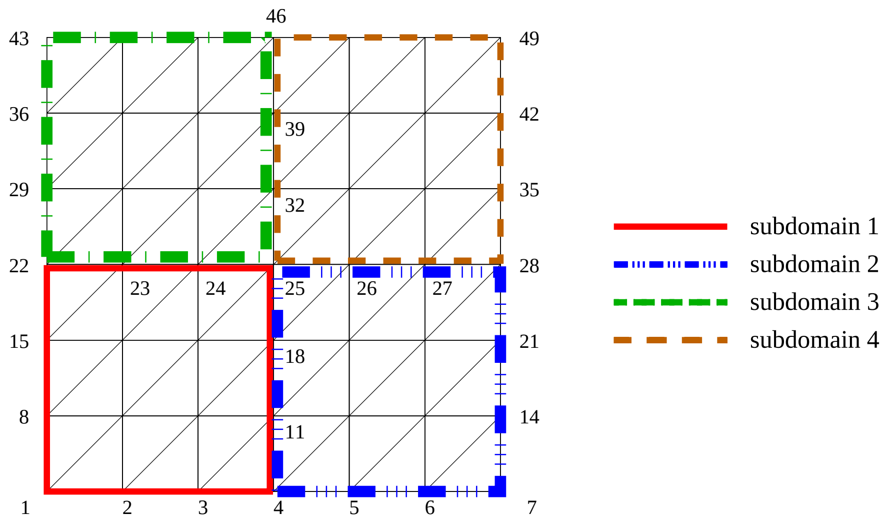

Figure 3. Non-overlapping subdomains with triangulation and row-wise numbering.

4.1 Communication groups and node sets

Besides the global inner products in PCG, any other communication is only next neighbour communication involving a subset of all processors. The model example in Fig. 3 possesses by definition (9) the communication groups $\{1, 2, 3, 4\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 4\}$, $\{3, 4\}$ and trivially $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$. These communication groups σ_k are uniquely ordered with the rules

- $|\sigma_k| > |\sigma_\ell| \quad \Rightarrow \quad k > \ell$,
- $|\sigma_k| = |\sigma_\ell|$ and σ_k lexicographically smaller than $\sigma_\ell \quad \Rightarrow \quad k > \ell$.

This ordering guarantees the consistency of local and global ordering and prevents communication deadlocks in the algorithm.
Let mc_s denote the number of communication groups of subdomain Ω_s. Then we have $mc_1 = 4$ in our example where $\sigma_1 = \{1, 2, 3, 4\}$, $\sigma_2 = \{1, 2\}$, $\sigma_3 = \{1, 3\}$ and $\sigma_4 = \{1\}$. The last one does not require any communication but simplifies the algorithms. It is obvious from Fig. 3, that $\sigma_1 = \sigma^{[25]}$, $\sigma_2 = \sigma^{[18]}$, $\sigma_3 = \sigma^{[24]}$ and $\sigma_4 = \sigma^{[1]}$. This leads directly to the idea to group locally the nodes ω_s of subdomain Ω_s into four classes with respect to a communication group σ_k :

- active nodes: $\omega_a := \{i \in \omega_s \,:\, \sigma^{[i]} \equiv \sigma_k\}$,
- visible nodes: $\omega_v := \{i \in \omega_s \,:\, \sigma^{[i]} \subset \sigma_k\}$,
- grouped nodes: $\omega_g := \{i \in \omega_s \,:\, \sigma^{[i]} \supset \sigma_k\}$,
- invisible nodes : $\omega_s \setminus (\omega_a \cap \omega_v \cap \omega_g)$.

This results, e.g., for subdomain Ω_1 and $\sigma_3 = \{1, 3\}$ in $\omega_a = \{22, 23, 24\}$, $\omega_g = \{25\}$ and $\omega_v = \{1, 2, 3, 8, 9, 10, 15, 16, 17\}$.

We need the accumulated stiffness matrix instead of the given distributed one for the determination of strong connections as well as for the calculation of interpolation weights. A first step to control the admissible interpolation pattern consists in a special accumulated matrix

$$\widetilde{\mathfrak{K}} = \sum_{s=1}^{P} \mathcal{A}_s^T \widetilde{\mathsf{K}}_s \mathcal{A}_s$$

$$\text{with} \qquad \widetilde{\mathsf{K}}_s^{[i,j]} = \begin{cases} \mathsf{K}_s^{[i,j]} & \text{iff} \quad \sigma^{[i]} \subseteq \sigma^{[j]} \vee \sigma^{[i]} \supseteq \sigma^{[j]} \\ 0 & \text{else} \end{cases} \quad , \tag{22}$$

which is locally stored as $\widetilde{\mathfrak{K}}_s = \mathcal{A}_s \widetilde{\mathfrak{K}} \mathcal{A}_s^T$. This means for Fig. 3, that the resulting matrix $\widetilde{\mathfrak{K}}$ does not contain entries $K^{[18,26]}$, $K^{[24,32]}$ and their transposed entries.

The second step takes advantage of the fact that our ordering of the communication groups provides already the right relation between the nodes due to to ω_a, ω_v and ω_g. The local coarsening will be done subsequently on the active sets of all communication groups σ_k. We have to guarantee coherency of the resulting coarse and fine nodes for active sets shared by more than one subdomain. The easiest way consists in a coarsening only on the root process of communication group σ_k, broadcast the result to the group and do the coarsening with the marked coarse nodes ω_m on the remaining processes. The two coarsening parts before and after the communication can be handled by the same routine $\textsc{CoarseP}(\{S^i\}, \omega_m, \omega_a, \omega_v)$, which can be derived easily from the sequential routine $\textsc{Coarse}$ on page 5. The sequential routine is now a special case of the parallel one and corresponds to the call $\textsc{CoarseP}(\{S^i\}, \emptyset, \omega_h, \omega_h)$. A third step has to restrict the strong connections such that no fine node from the active set ω_a has connections to the visible nodes from ω_v.

These three steps guarantee the admissible pattern (20) for interpolation and restriction. Algorithm 4 presents the routine for parallel coarsening on each subdomain Ω_s.

The modified strong connections $\{S^i\}$ ensure together with $\widetilde{\mathfrak{K}}$ the admissible matrix pattern for the interpolation. Therefore, we call

$$\mathfrak{P}_s \leftarrow \textsc{Weights}(\{S^i\}, \widetilde{\mathfrak{K}}, \omega_C, \omega_F)$$

on each processor s without any communication.

Algorithm 4 Parallel coarsening $(\{S^i\}, \omega_C, \omega_F) \leftarrow \textsc{ParCoarse}(\{S^i\}, \omega_s)$

Determine list $[\sigma_k]_{k=1,\ldots,\mathrm{mc}_s}$ (= communicator groups).
for all $k = 1, \ldots, \mathrm{mc}_s$ **do**
  $\omega_a \leftarrow \{i \in \omega_s \, : \, \sigma^{[i]} \equiv \sigma_k\}$
  $\omega_v \leftarrow \{i \in \omega_s \, : \, \sigma^{[i]} \subset \sigma_k\}$
  $\omega_g \leftarrow \{i \in \omega_s \, : \, \sigma^{[i]} \supset \sigma_k\}$
  $\omega_m \leftarrow \emptyset$
  if $s ==\textsc{Root}(\sigma_k)$ **then**
    $(\omega_C, \omega_F) \leftarrow \textsc{CoarseP}\ (\{S^i\}, \omega_m, \omega_a, \omega_v)$
  end if
  $\omega_m \leftarrow \textsc{Broadcast}(\sigma_k, \omega_C \cap \omega_a)$
  if $s! =\textsc{Root}(\sigma_k)$ **then**
    $(\omega_C, \omega_F) \leftarrow \textsc{CoarseP}\ (\{S^i\}, \omega_m, \omega_a, \omega_v)$
  end if
  for all $i \in \omega_a$ **do**
    if $i \in \omega_F$ **then**
      $S^i \leftarrow S^i \cap (\omega_a \cup \omega_g)$
    else
      $S^i \leftarrow S^i \cap (\omega_a \cup \omega_v)$
    end if
  end for
end for

4.3 Coarse grid matrix and parallel setup

The pattern condition (20) for the interpolation matrix $\mathfrak{P}$ is fulfilled by construction in Alg. 4. Therefore, we can apply (21) with the distributed fine grid matrix K so that the coarse matrix K_H is again distributed stored and can be calculated fully in parallel without communication, i.e., we call

$$\mathsf{K}_{H,s} \leftarrow \textsc{Galerkin}(\mathsf{K}_s, \mathfrak{P}_s)$$

locally on each processor s.

The parallel setup on page 14 collects the subroutines of previous sections. The coarsening part in Alg. 5 terminates if the global number of coarse nodes becomes smaller than CoarseGrid, cf. the sequential part.

4.4 Object oriented approach for the sequential and parallel code

The implementation of the parallel code exploits the C++ principles of overloading and inheritance. Given a class hierarchy with several types (e.g. SparseMatrix, BlockMatrix) derived from the base class (e.g. BaseMa-

Algorithm 5 Parallel Setup $\textsc{ParSetup}(\mathsf{K}_h, \omega, \ell)$

if $|\omega| > \textsc{CoarseGrid}$ **then**
  for all $s = 1, \ldots, P$ **do**
    $\mathsf{K}_s^1 \leftarrow \mathsf{K}_s$, $\omega_s^1 \leftarrow \omega_s$,
    $\widetilde{\mathfrak{K}} \leftarrow \left(\sum_{i=1}^{P} \mathcal{A}_i^T \widetilde{\mathsf{K}}_i \mathcal{A}_i\right)_{\text{pattern}}$, $\widetilde{\mathfrak{K}}_s \leftarrow \mathcal{A}_s \widetilde{\mathfrak{K}} \mathcal{A}_s^T$
    $S^i \leftarrow \textsc{GetStrong}(\widetilde{\mathfrak{K}}_s) \qquad \forall i \in \omega_s$
    $\omega_C \leftarrow \emptyset$, $\omega_F \leftarrow \emptyset$
    $(\{S^i\}^\ell, \omega_{C,s}, \omega_{F,s}) \leftarrow \textsc{ParCoarse}(\{S^i\}^\ell, \omega_s^\ell)$
    $\mathfrak{P}_s^\ell \leftarrow \textsc{Weights}(\{S^i\}^\ell, \widetilde{\mathfrak{K}}, \omega_{C,s}, \omega_{F,s})$, **free**$(\{S^i\}^\ell, \widetilde{\mathfrak{K}})$
    $\mathsf{K}_s^{\ell+1} \leftarrow 0$
    $\mathsf{K}_s^{\ell+1} \leftarrow \textsc{Galerkin}(\mathsf{K}_s^\ell, \mathfrak{P}_s^\ell)$
    $\omega_s^\ell \leftarrow \omega_{C,s}$, $\ell \leftarrow \ell + 1$
    $\textsc{ParSetup}(K^{\ell+1}, \omega^{\ell+1}, \ell + 1)$
  end for
else
  $\textsc{CoarseLevel} \leftarrow \ell$
end if

TRIX), we introduce an additional "parallel" class (e.g. PARALLELMATRIX) derived from the base class. The parallel class has a reference to some ob-

```
// Class hierarchy: BaseMatrix, SparseMatrix, ParallelMatrix
class BaseMatrix
      virtual double GetDiag(int i) const = 0;
      virtual BaseMatrix & GetMatrix() { return *this; };

class SparseMatrix :: BaseMatrix
      virtual double GetDiag(int i) const {return Get(i,i); };

class ParallelMatrix :: BaseMatrix
      BaseMatrix * matrix;
      Vector accumulated_diag;
      Accumulator * accumulator;
      ParallelMatrix(BaseMatrix * am) { matrix = am; };
      virtual double GetDiag(int i) const
           { return accumulated_diag.Get(i); };
      virtual BaseMatrix & GetMatrix() { return *matrix;};
```

ject of the base class and has access to additional "parallel" data structures. The method GETOBJECT() (e.g. GETMATRIX()) of the base class returns the reference of the object itself. For the parallel class, this method returns the reference of the "sequential" object. This concept allows to switch easily between the seqential and parallel methods depending on the requirements of the algorithm. For example,

$$\textsc{GetDiag}(i)$$

is to return the accumulated diagonal element of row i as required for the Jacobi iteration within the smoother. On the other hand,

$$\text{GetMatrix}().\text{GetDiag}(i)$$

accesses the diagonal element of the distributed matrix.

Using this approach, the changes in the sequential code are minimal as long as the existing code is well structured and global methods, including input and output, have already been implemented as methods of appropriate class variables. Of course, additional parallel-specific code sequences have to be added. This includes the interfaces to the parallel library DDComm [15] for the setup of the communication. In particular, access to the topology of the mesh has to be provided. Furthermore, the methods which are to be overloaded have to be supplied. In the example given above, this includes the initialization of the accumulated diagonal:

```
for(i=1; i<=n; i++)
     accumulated_diag.Elem(i) = matrix->Get(i,i);
accumulator.MultInline(accumulated_diag);
```

The diagonal is being accummulated using the methods of the Distributed Data Communication library DDComm.

5 Numerical results

Let us consider the Poisson problem $\Delta u = 1$ in Ω and $u = 0$ on $\partial\Omega$ with some Lipschitz domain Ω. This problem reads in variational form: Find $u \in H_0^1(\Omega)$ such that

$$\int_\Omega \nabla u \, \nabla v \, dx = \int_\Omega 1 \, v \, dx \quad \forall v \in H_0^1(\Omega).$$

In particular we consider the geometry of a simplified crank shaft (see Fig. 4) and our model problem corresponds to a heat conduction problem. The use of the mesh generator NETGEN [23] results in a tetrahedral mesh shown in Fig. 4. The mesh being shown consists of 11029 elements. A partition of this mesh is computed by the modified recursive spectral bisection method with interface smoothing [8]. This algorithm produces an almost uniform distribution of the elements to the subdomains for any number of subdomains. For the calculations presented below, the mesh has been refined 4 times uniformly after the partitioning resulting in globally 3227648 elements and 576833 nodes. Hence, using linear Lagrange finite elements we are faced with 576833 global unknowns.

Refinement, matrix generation and iterative solver are driven by FEPP [16] whereas the AMG preconditioner is realized within PEBBLES [20]. Both modules use the Distributed Data Communication Library DDComm [15].

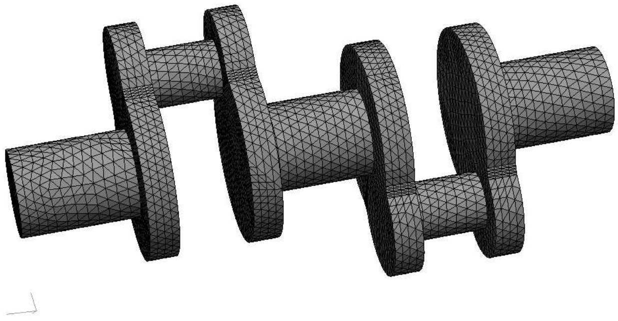

Figure 4. The mesh with 50432 elements and 11029 nodes.

As described before, we solve the resulting matrix equation by the AMG-PCG solver (Alg. 2 together with Alg. 3). In particular, we apply AMG with element preconditioning [19]. The time required for generating spectrally equivalent M-(element-)matrices is contained in the matrix generation time given below. The interpolation weights are chosen by $1/|S^i|$, i.e., the fine grid value is obtained by averaging the values of connected coarse grid nodes. We apply a V-cycle with 2 pre- and post-smoothing steps and a direct coarse grid solver. The smoother is a block Jacobi smoother with Gauss-Seidel smoothing in blocks containing interior unknowns of the subdomains. We choose a relative accuracy of 10^{-4} as stopping criterion.

First we run the experiments on a SGI ORIGIN 2000 with R12000, 300MHz processors and overall 20 GB of main memory. We investigate the speedup for 1 up to 32 processors. Table 1 shows the number of iterations, the wall clock-time (measured by MPI_WTIME) required for the AMG setup, matrix generation and solver. The time for grid refinement and partitioning as well as for the communication setup is not included. Since both, coarsening and smoother depend on the decomposition into subdomains, we observe that the number of iterations depends on the number of processors. That is why we have added the speedup with respect to 1 Iteration to the corresponding speedup results in Table 2 and in Fig. 5. The matrix generation is purely local and gives the reference curve for the quasi optimal speedup. Looking at the speedup of the setup phase, we observe some kind of saturation effects which are due to the increasing communication effort. In contrast to the solver, these communication steps have to be synchronized. Hence, increasing P for a fixed problem size, increases the sequential overhead. However, in many applications, e.g. transient or nonlinear problems, a fixed setup can be used for repeated calls of the solver. Then the speedups for the solver alone are of interest. Here we observe optimal speedups. Moreover we profit from cache effects since the local problem size decreases.

P	It.	Matrix	Setup	Solver	All
1	31	211.9	184.2	383.4	779.3
2	33	106.5	94.5	223.7	424.7
4	33	51.0	50.4	111.5	219.9
8	24	24.6	22.5	29.3	76.4
16	25	12.6	12.2	13.3	38.1
24	25	9.0	10.2	9.5	28.7
32	26	7.3	8.4	7.0	22.7

Table 1
Number of processors (P), number of iterations (It.), wall-clock time for the matrix generation (Matrix), the setup phase (Setup), the solver (Solver) and the sum of times (All) in seconds.

P	1 It.	Matrix	Setup	Solver	All
1	1.0	1.0	1.0	1.0	1.0
2	1.8	2.0	1.9	1.7	1.8
4	3.7	4.1	3.6	3.4	3.5
8	10.1	8.6	8.2	13.1	10.2
16	23.2	16.8	15.1	28.8	20.4
24	32.5	23.5	18.0	40.3	27.2
32	45.9	29.0	21.9	54.8	34.3

Table 2
Speedups for one iteration of the solver (1 It.), the matrix generation (Matrix), the setup phase (Setup), the solver (Solver) and the sum of times (All) for $P = 1 \ldots 32$.

Now we repeat our experiments on a PC cluster with 4 double processor machines with INTEL Pentium 500MHz processors and at least 256 KB of main memory. The machines are running under SUSE LINUX 6.3 and are connected via fast Ethernet. The LAM-distribution of MPI is used. Because of memory limitations we reduce the problem size. In particular 3 refinement steps give 403456 elements and 77153 nodes. The results are presented in the same way as above in Table 3. We consider speedup for 1 up to 8 processors in Table 4 and Fig. 6. As expected, we observe lower efficiency compared to the computaions on the ORIGIN 2000. However, the PC cluster caused no addtional costs for the network since the standard fast Ethernet has been used.

Table 5 shows the overall time on the ORIGIN 2000 and the PC cluster for the same (smaller) problem size. For very few processors the faster processors of the PC cluster give lower computaion time. If the effort for communications

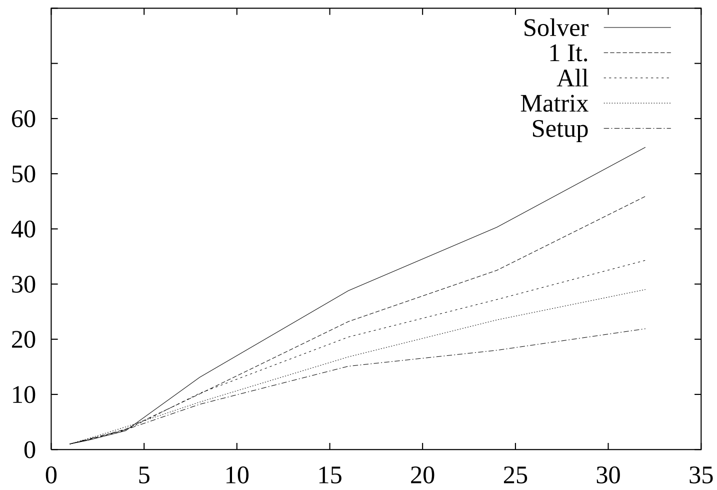

Figure 5. Speedup results on the SGI ORIGIN 2000.

P	It.	Matrix	Setup	Solver	All
1	23	20.0	10.4	24.0	54.4
2	23	9.9	5.4	12.0	27.3
4	23	5.0	3.4	6.4	14.8
6	22	3.4	2.5	4.9	10.8
8	18	2.5	2.3	3.4	8.2

Table 3
Number of processors (P), number of iterations (It.), wall-clock time for the matrix generation (Matrix), the setup phase (Setup), the solver (Solver) and the sum of times (All) in seconds.

P	1 It.	Matrix	Setup	Solver	All
1	1.0	1.0	1.0	1.0	1.0
2	2.0	2.0	1.9	2.0	2.0
4	3.7	4.0	3.0	3.7	3.7
6	4.7	5.9	4.1	4.9	5.0
8	5.5	8.0	4.5	7.1	6.6

Table 4
Speedups for one iteration of the solver (1 It.), the matrix generation (Matrix), the setup phase (Setup), the solver (Solver) and the sum of times (All) for $P = 1 \ldots 8$.

grows with P, the ORIGIN 2000 outperforms the PC Cluster due to the superior network.

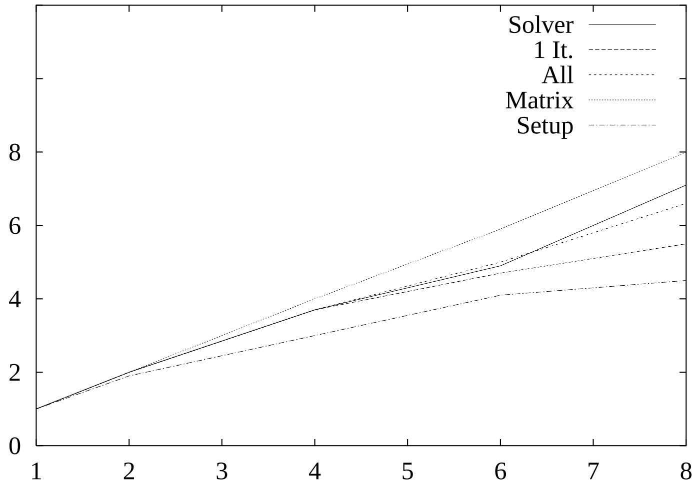

Figure 6. Speedup results on a PC cluster.

P	ORIGIN 2000	PC Cluster
1	59.6	54.4
2	31.1	27.3
4	15.6	14.8
8	7.1	8.2

Table 5
Sum of times on the ORIGIN 2000 and the PC Cluster.

6 Conclusions

We have presented an approach for the design and implementation of parallel AMG for distributed memory computers. Although results have only been shown for scalar problems, the algorithm can be applied directly to block-systems, arising, e.g., in linear elasticity. Moreover, the same concept of parallelization can be applied to magnetic field problems using Nédélec finite elements, i.e., edge-based degrees of freedom [9]. Together with the AMG-specific ideas developed in [21], a parallel AMG solver for 3D Maxwell equations can be realized immediately.

References

[1] D. Braess. Towards algebraic multigrid for elliptic problems of second order. *Computing*, 55:379–393, 1995.

[2] C. C. Douglas and M. B. Douglas. MGNet Bibliography. Department of Mathematics and the Center for Computational Sciences, University of

Kentucky, Lexington, KY, USA and Department of Computer Science, Yale University, New Haven, CT, USA, 1991–1999 (last modified on November 6, 1999); see http://www.mgnet.org/bib/mgnet.bib.

[3] R. Falgout, V. E. Henson, J. E. Jones, and U. M. Yang. Boomer AMG: A parallel implementation of algebraic multigrid. Techn. Report UCRL-MI-133583, Lawrence Livermore National Laboratory, March 1999.

[4] M. Griebel and G. Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Computing*, 25:827–843, 1999.

[5] G. Haase. Parallel incomplete cholesky preconditioners based on the non-overlapping data distribution. *Parallel Computing*, 24(11):1685–1703, 1998.

[6] G. Haase. *Parallelization of Numerical Algorithms for PDEs.* Teubner, 1999. in German.

[7] G. Haase. A parallel AMG for overlapping and non-overlapping domain decomposition. *Electronic Transactions on Numerical Analysis (ETNA)*, 10:41–55, Feb. 2000.

[8] G. Haase and M. Kuhn. Preprocessing for 2D FE-BE domain decomposition methods. *Comp. Vis. Sci.*, 2:25–35, 1999.

[9] G. Haase, M. Kuhn, and U. Langer. Parallel multigrid 3D Maxwell solvers. In *Proceedings of the* 12^{th} *International Conference on Domain Decomposition in Chiba, October 1999.* DDM.ORG, 2000. accepted for publication.

[10] G. Haase, U. Langer, and A. Meyer. The approximate Dirichlet decomposition method. part I,II. *Computing*, 47:137–167, 1991.

[11] G. Haase, U. Langer, S. Reitzinger, and J. Schöberl. A general approach to AMG. Technical report, SFB F013, 2000. in preparation.

[12] M. Jung. On the parallelization of multi-grid methods using a non-overlapping domain decomposition data structure. *Applied Numerical Mathematics*, 23(1):119–137, 1997.

[13] F. Kickinger. Algebraic multigrid for discrete elliptic second-order problems. Technical Report 96-5, Johannes Kepler Universität Linz, Institut für Mathematik, 1997. www.nathan.numa.uni-linz.ac.at/pub/technical_reports/.

[14] A. Krechel and K. Stüben. Parallel algebraic multigrid based on subdomain blocking. *Parallel Computing*, 2000. Submitted.

[15] M. Kuhn. DDComm – Distributed Data Communication library. SFB "Numerical and Symbolic Scientific Computing", http://www.sfb013.uni-linz.ac.at., 2000.

[16] M. Kuhn, U. Langer, and J. Schöberl. Scientific computing tools for 3d magnetic field problems. In J. Whiteman, editor, *The Mathematics of Finite Elements and Applications.* John Wiley & Sons, 2000. See also http://www.sfb013.uni-linz.ac.at.

[17] K. H. Law. A parallel finite element solution method. *Computer and Structures*, 23(6):845–858, 1989.

[18] A. Meyer. A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain. *Computing*, 45:217–234, 1990.

[19] S. Reitzinger. Algebraic Multigrid and Element Preconditioning II. Technical Report 99-18, SFB F013 "Numerical and Symbolic Scientific Computing", Johannes Kepler University Linz, 1999. Submitted.

[20] S. Reitzinger. PEBBLES – user's guide. SFB "Numerical and Symbolic Scientific Computing", http://www.sfb013.uni-linz.ac.at., 1999.

[21] S. Reitzinger and J. Schöberl. Algebraic multigrid for edge elements. Technical Report 00-15, SFB F013 "Numerical and Symbolic Scientific Computing", Johannes Kepler University Linz, 2000.

[22] J. W. Ruge and K. Stüben. Algebraic multigrid (AMG). In S. McCormick, editor, *Multigrid Methods*, volume 5 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, 1986.

[23] J. Schöberl. NETGEN - An advancing front 2D/3D-mesh generator based on abstract rules. *Comput.Visual.Sci*, 1:41–52, 1997.

[24] P. Vanek, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56:179–196, 1996.