UCRL-TR-210788

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Scalable Parallel Algebraic Multigrid Solvers

Randolph Bank, Shaoying Lu, Charles Tong,
Panayot Vassilevski

March 24, 2005

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

SCALABLE PARALLEL ALGEBRAIC MULTIGRID SOLVERS

RANDOLPH E. BANK [*], SHAOYING LU [†], CHARLES TONG [‡], AND PANAYOT S. VASSILEVSKI [§]

Abstract. We propose a parallel algebraic multilevel algorithm (AMG), which has the novel feature that the subproblem residing in each processor is defined over the entire partition domain, although the vast majority of unknowns for each subproblem are associated with the partition owned by the corresponding processor. This feature ensures that a global coarse description of the problem is contained within each of the subproblems. The advantages of this approach are that interprocessor communication is minimized in the solution process while an optimal order of convergence rate is preserved; and the speed of local subproblem solvers can be maximized using the best existing sequential algebraic solvers.

Key words. Algebraic multigrid algorithm, Parallel efficiency, Domain decomposition, Bank–Holst algorithm.

AMS subject classifications. 65N50, 65N30

1. Introduction.

The numerical solution of linear systems arising from the discretization of partial differential equations(PDEs) is often the most computationally expensive part of scientific applications. On single-processor computers, the Krylov subspace methods with multigrid preconditioners have been shown to have optimal complexity in both memory and time consumption. Geometric multigrid (GMG) methods are very efficient for solving a large class of problems. However, in some cases, a GMG method may not be a good candidate for a solver or preconditioner. For example, the linear system could have no underlying hierarchy of grids; the finite element discretization could have very complex geometry, so the coarsest problem is too large to solve by a direct solver or a classical iterative method; constructing an efficient smoothing operator for 3D anisotropic problems is not an easy task. With the potential of overcoming these difficulties, algebraic multigrid (AMG) methods are of particular interest. Another reason the AMG method is popular is because it can be used as a "black box" solver, requiring only a matrix and a right hand side vector as input.

The AMG method was introduced in the 1980's [8, 9, 10, 11] and developed into a fairly general algorithm immediately afterward [24, 25, 26, 28]. There had been no substantial development in this field until the mid-1990's, when a strong increase of interest in AMG methods occurred [7, 12, 13, 14, 15, 23, 27, 29]. Some applications of AMG methods can be found in [18, 21, 22]. Sequential AMG methods differ in the way

[*]Department of Mathematics University of California, San Diego La Jolla, California 92093-0112. email:rbank@ucsd.edu. The work of this author was supported by the National Science Foundation under contract DMS-0208449, and through the UCRP program at Lawrence Livermore National Laboratory. The UCSD Scicomp Beowulf cluster was built using funds provided by the National Science Foundation through SCREMS Grant 0112413, with matching funds from the University of California at San Diego.

[†]Department of Mathematics University of California, San Diego La Jolla, California 92093-0112. email:sylu@math.ucsd.edu. The work of this author was supported by the National Science Foundation under contract DMS-0208449, and through the UCRP program at Lawrence Livermore National Laboratory.

[‡]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, email: chtong@llnl.gov

[§]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, email: vassilevski1@llnl.gov the interpolation operators and the hierarchy of stiffness matrices are constructed. In the literature, Ruge and Stüben [25] give a review of the classical AMG algorithms; Stüben [27] is an updated review of AMG methods for people with background in GMG methods. Several parallel algebraic multigrid algorithms have been developed for today's large scale parallel computers [16, 17, 19, 30]. Great effort has been devoted to improving parallelization efficiency, or scalability, of these methods. A major obstacle for the scalability of multilevel PDE solvers is the inter-processor communication cost, especially at coarser levels.

In this paper we propose a parallel algebraic multilevel algorithm for use as a solver or as a preconditioner in conjunction with an iterative method. This parallel multilevel solver differs from traditional parallel solvers in that each subproblem by itself is a specially coarsened version of the global problem, which contains unknowns from all partitions. On each processor, the local problem consists of fine unknowns belonging to its own partition and coarse unknowns from all other partitions. In other words, the global problem is coarsened outside a given partition to produce a local problem. Here we note that the terminology, "local", is used throughout this paper to refer to a subject that is stored on the current processor. A *local* problem is a problem stored on the current processor. It may contain unknowns from neighboring partitions; it may even contain unknowns from all partitions. On the other hand, a *global* problem is a problem stored across all processors, which always contains unknowns from all partitions.

Our algorithm is designed to reduce the global communication cost during iterative solving steps. We solve the global problem by solving a partially-fine, partially-coarse local problem on each processor, as a preconditioner. Since each local problem is stored completely on a single processor, no communication is required within one iteration. A direct motivation of our algorithm is the geometric parallel multigrid algorithms described in [2, 3, 4, 5, 6], where the algorithms are proposed for solving partial differential equations using parallel adaptive meshing scheme. The convergence of this type of algorithms is backed by the multigrid convergence theory: high frequency error is resolved by locally fine unknowns; low frequency error is resolved by globally coarse unknowns.

In section 2, we will describe the algorithm of constructing two-level and three-level parallel preconditioners. Local interpolation operators will be derived from a global interpolation operator. Using various two-level and three-level interpolation operators, local stiffness matrices and vectors will be computed. Local problems are solved on each processor and local solutions are collected to obtain a global update of the solution. In section 3 we will discuss parallel implementation details in terms of matrix and vector data structure. The algorithms for building various local stiffness matrices and vectors will be presented, algorithm complexity will be discussed. In section 4, we present numerical results on scalability, in iteration counts and solution time. We study the three-level solver with and without a two-level coarser solver, in comparison with BoomerAMG, see [1] [17]. Our implementation is built on top of the HYPRE library developed in the Center for Applied Scientific Computing of the Lawrence Livermore National Laboratory. The global interpolation operators are generated using parallel an algebraic coarsening scheme implemented in BoomerAMG.

2. Parallel AMG Algorithms.

Both classic AMG and GMG algorithms consist of two phases: initialization and iterative solving. In the initialization phase, a hierarchy of interpolation matrices and stiffness matrices are generated. In the solving phase, high frequency errors and low frequency errors are reduced by the interaction of smoothing and coarse grid correction. In GMG, since the interpolation operators are determined by the underlying geometry, an appropriate smoothing operator must be chosen to produce geometrically smooth error. AMG differs from GMG in that no geometric grids are given. As a result, a good interpolation operator can be constructed such that the smooth error is well represented by the coarse subspace; the smoothing operator may be as simple as a Gauss-Seidel iteration.

Existing parallel AMG methods distribute the work load of initialization and iterative solution among different processors on a parallel cluster, with a communicate-when-necessary approach. First, with p processors, the unknowns of the linear system $Ax = f$ is divided into p partitions; each partition is assigned to one processor. Second, the interpolation operators I_{l+1}^{l} are generated at all levels by coarsening the matrix A in parallel. I_{l+1}^{l} maps unknowns from the level $l+1$ to a coarser level l; the level 0 is the finest. The stiffness matrices $A_{l+1} = (I_{l+1}^{l})^t A_l I_{l+1}^{l}$ are generated at all levels as well. Finally, the problem is solved using a standard multilevel scheme. In this type of algorithm, both A_l and I_{l+1}^{l} are parallel matrices. A parallel matrix is typically stored row-wise or column-wise. Each processor owns the rows/columns corresponding to its own partition. As a result, each matrix-vector multiplication requires inter-processor communication. Communication may be the dominant cost at coarser levels, where each processor contains only a few unknowns.

Our proposed parallel AMG algorithms try to reduce the communication complexity of the solving stage by restricting the communication to no more than two levels of the multigrid hierarchy. It differs from the existing parallel AMG methods in the way that coarser level stiffness matrices are constructed and stored. We coarsen the global stiffness matrix only outside the current partition, and store the partially coarsened matrix completely on the current processor. At a fixed level l, we denote the interpolation operator, between the level l and the finest level, by P. P is a global parallel matrix which is the product of l global interpolation operators $I_1^0 I_2^1 \cdots I_l^{l-1}$. The local stiffness matrix A_p on the processor p is then computed using P. A_p consists of fine unknowns from the partition owned by the current processor, and coarse unknowns at level l, from the partitions owned by the remaining processors. This local problem can then be solved by a direct solver or a sequential multilevel solver. Since A_p is a sequential matrix stored completely on processor p, the solution of a local problem does not require any communication.

The unknowns contained in the local problems of our proposed two-level and three-level algorithms are illustrated in figure 2.1, where the global unknowns are divided into 25 partitions. For two-level algorithms, the local stiffness matrix A_p consists of fine unknowns from the partition p and coarse unknowns from all other partitions. In three-level algorithms, the local stiffness matrix A_p consists of fine unknowns from the partition p and coarse unknowns from only the neighboring partitions. The unknowns from other partitions are discarded.

Comparing to an existing parallel AMG algorithm, our proposed algorithm will consume more time and memory in its initialization phase, due to the fact that the local stiffness matrix A_p has to be formed after the usual AMG initialization. However, we expect our algorithm to be more efficient during the solution phase since less communication is required at each iteration. This property makes our algorithm especially attractive for problems where one initialization stage is followed by repeated solving steps. For example, it is well suited for solving time dependent problems, where the same stiffness matrix is used for every time step.

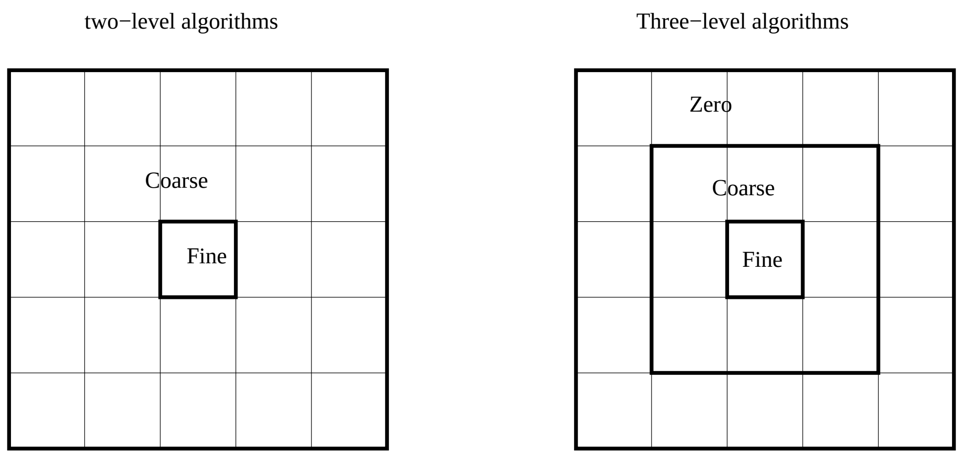

FIG. 2.1. *Unknowns contained in two-level and three-level algorithms*

2.1. Interpolation Matrix P. The global interpolation matrix P is a parallel matrix. P can be written as

$$P = \begin{bmatrix} P_{cc} & P_{co} \\ P_{oc} & P_{oo} \end{bmatrix},$$

where P_{cc} is associated with unknowns in the current partition. P_{co} connects fine unknowns in the current partition to coarse unknowns from other partitions. On the other hand, P_{oc} describes how the fine unknowns from other partitions depend on the coarse unknowns in the current partition. P_{oo} is associated with the unknowns outside the partition p.

A one dimensional example with geometric interpretation is given below. Let the matrix P be:

$$P = \left[\begin{array}{cccc|ccc}
\vdots & \vdots & & & & & \\
 & 1.0 & & & & & \\
 & 0.75 & 0.25 & & & & \\
 & 0.5 & 0.5 & & & & \\
 & 0.25 & 0.75 & & & & \\
 & & 1.0 & & & & \\
 & & & 1.0 & & & \\
\hline
 & & & 0.75 & 0.25 & & \\
 & & & 0.5 & 0.5 & & \\
 & & & 0.25 & 0.75 & & \\
 & & & & 1.0 & & \\
 & & & & 0.75 & 0.25 & \\
 & & & & 0.5 & 0.5 & \\
 & & & & 0.25 & 0.75 & \\
 & & & & & 1.0 & \\
 & & & & & \vdots & \vdots
\end{array}\right]. \qquad (2.1)$$

An underlying 1-d mesh is shown in figure 2.2. The mesh is partitioned into two

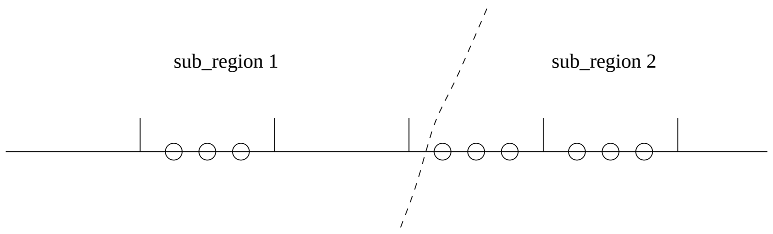

FIG. 2.2. *One dimensional grid coarsened*

subregions by a dashed line. Coarse grid points are represented by bars and fine grid points by circles. As shown in figure 2.2, as well as in equation 2.1, the fine unknowns of the first subregion are not connected to the second subregion, while three fine unknowns in the second subregion depend on a coarse unknown in the first subregion.

2.2. Two-level Coarsening Algorithm . In this subsection we describe several two-level coarsening algorithms, to compute the local stiffness matrix A_p from the interpolation operator P. First, we construct a local interpolation matrices π_p. On processor p, the matrix π_p will be used to compute A_p,

$$A_p = \pi_p^t A \pi_p.$$

Since π_p is also an interpolation operator, it is required to satisfy the following criteria.

1. The entries of π_p have values between 0 and 1.
2. The row sums of the matrix are 1.
3. Since the set of coarse unknowns is a subset of the unknowns of the original linear system, π_p has one and only one entry 1 at each column.

The matrix P partitioned as

$$P = \begin{bmatrix} P_{cc} & P_{co} \\ P_{oc} & P_{oo} \end{bmatrix} = \begin{bmatrix} \bar{P}_c \\ \bar{P}_o \end{bmatrix}, \tag{2.2}$$

we describe three variations of the interpolation matrix π_p. All of them restrict coarsening to the unknowns outside the current partition, by substituting $\begin{bmatrix} I & 0 \end{bmatrix}$ for $\bar{P}_c$ in P. They differ in the way that the connections between fine unknowns outside and coarse unknowns inside partition p are treated.

1. This connection is completely ignored by eliminating the block P_{oc}.

$$\pi_p = \begin{bmatrix} I & 0 \\ 0 & P_{oo} \end{bmatrix}. \tag{2.3}$$

2. Alternatively, let

$$\pi_p = \begin{bmatrix} I & 0 \\ 0 & \tilde{P}_o \end{bmatrix}, \tag{2.4}$$

   where $\tilde{P}_o$ consists of the nonzero columns of $\bar{P}_o$. The connection is cut off implicitly by duplicating the unknowns inside partition p, which influence the unknown outside this partition. It seems that equation 2.4 will yield an algorithm which converges faster than that derived from equation 2.3, because the information in P_{oc} is retained. However, it has the potential of producing a singular matrix A_p, since this formulation duplicates the unknowns on the interface of partition p.

3. Let

$$\pi_p = \begin{bmatrix} I & 0 \\ \bar{P}_{oc} & P_{oo} \end{bmatrix}, \tag{2.5}$$

   where $\bar{P}_{oc} = P_{oc}T$, where T maps coarse unknowns to their globally fine indices. This choice of π_p combines the good qualities of both 2.3 and 2.4. This algorithm converges fast; it always produces a nonsingular local stiffness matrix. However, the underlying algorithm is expensive to implement, within the parallel data structures of HYPRE. Nevertheless, it is easy to implement the algorithm in the context of our three-level algorithms.

Once the local interpolation matrix π_p is chosen, the parallel AMG algorithm can be defined.

ALGORITHM 2.1. *(Solve linear system $Au = f$, initial guess $u = u_0$.)*

```
• Initialization:
    1. generate matrix $A_p = \pi_p^t A \pi_p$;
    2. compute initial residual $r = f - A * u_0$.
• Solve (iterates until converges):
    1. compute local right hand side $r_p = \pi_p^t r$;
    2. solve local problem $A_p u_p = r_p$;
    3. form global update $du = R_p u_p$, where $R_p$ restricts $u_p$ to fine unknowns
       owned by processor $p$;
    4. update solution $u$ and right hand side $r$.
```

Given the interpolation matrix P_l at a coarsening levels l, the matrices π_p can be constructed to compute the local problems $A_p u_p = r_p$. The convergence rate of the two-level algorithms improves as the coarsening level l decreases. On the other hand, because more unknowns are present in the local problems, the memory complexity and the operator complexity increases, especially when there is a large number of processors.

2.3. Three-level Algorithms. The content of the local stiffness matrix A_p, corresponding to the three-level algorithms is shown in figure 2.1. The matrix consists of fine unknowns in partition p, coarse unknowns in the neighboring partitions and no unknowns elsewhere. In principle, based on any of the two-level algorithms described in 2.2, we can construct a three-level local interpolation matrix by cutting off the out-layer of unknowns from a two-level local interpolation matrix. However, since equation 2.4 is very likely to produce a singular local stiffness matrix, it will not be chosen to generate a three-level interpolation operator. In practice, we consider only equations 2.3 and 2.5 as candidates for constructing a three-level interpolation matrix.

To introduce the three-level algorithm, we write the global interpolation matrix as a 3×3 block matrix, where the unknowns are grouped by the current partition, the neighboring partitions and the other partitions.

$$P = \begin{bmatrix} P_{cc} & P_{cn} & P_{co} \\ P_{nc} & P_{nn} & P_{no} \\ P_{oc} & P_{on} & P_{oo} \end{bmatrix}.$$

The local interpolation matrix derived from equation 2.3 is

$$\pi_p = \begin{bmatrix} I & 0 & 0 \\ 0 & P_{nn} & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{2.6}$$

Similarly, the local interpolation matrix derived from equation 2.5 is

$$\pi_p = \begin{bmatrix} I & 0 & 0 \\ P_{nc}T & P_{nn} & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{2.7}$$

As in the two-level case, the three-level local interpolation matrix π_p can be constructed at any given level l. The convergence of the algorithm speeds up as l decreases. At the same time, the operator complexity increases. The difference is that, in the three-level algorithms, each partition has a fixed number of neighbors, which is independent of the number of processors. Hence we will have better control on the increase of the operator complexity, when l is reduced to obtain faster convergence.

The corresponding iterative algorithm is again given by algorithm 2.1. Since the local stiffness matrices corresponds to only the unknowns of the current and the neighboring partitions in this algorithm, we expect the iteration count to grow as a logarithmic function of the number of global unknowns. To improve the scalability, a global coarse solver will be required to resolve the global smooth error.

2.4. Overlapping Schwartz Method. Because the three-level algorithm requires a global coarse grid correction, we use an overlapping Schwartz method, with the local problems overlapping on the neighboring partitions. Here we propose to use one of the three-level algorithms as the local smoothing operator, with only mild coarsening on the neighboring partitions; we use one of the two-level algorithms in 2.2 as the global coarse grid correction, with more aggressive coarsening outside a given processor. The following algorithm describes the overlapping Schwartz method:

ALGORITHM 2.2. *(Solving $Au = f$ using initial guess $u = u_0$)*

- *Initialization*
  1. *Form the three-level local stiffness matrix, with the local interpolation matrix defined as in equation 2.4, at coarsening level l;*
  2. *form the two-level local stiffness matrix, with local interpolation matrix defined as in equations 2.3, or in 2.5, at coarsening level $2l$;*
  3. *compute the initial residual $r = f - Au_0$.*
- *Main iteration*
  1. *Solve for the update du, using the three-level local stiffness matrix and algorithm 2.1, at coarsening level l;*
  2. *update the solution u and the residual r;*
  3. *solve for du using the two-level local stiffness matrix and algorithm 2.1, at coarsening level $2l$;*
  4. *update the solution u and the residual r.*

3. Parallel Implementation. The parallel implementation is built on the HYPRE library developed in the Center for Applied Scientific Computing of Lawrence Livermore National Laboratory. The coarsening algorithm uses the parallel algebraic solver/preconditioner BoomerAMG to generate multiple levels of global interpolation matrices P_l and global stiffness matrices A_l, where l stands for the level of coarsening.

In sections 2.2 and 2.3, our proposed two-level and three-level algorithms were described together with the formulation of the local interpolation matrices π_p. In practice, the local interpolation matrices are formed explicitly only in the three-level algorithms. In the two-level algorithms, the local stiffness matrices A_p and the local residuals r_p are constructed directly from P_l and A_l, using π_p implicitly. We describe below the detailed algorithms for constructing A_p and r_p.

3.1. Two-level Algorithms. The specific parallel algorithm used to construct the local stiffness matrix and the local residual is strongly influenced by the parallel data structure used in the HYPRE library. In HYPRE, a parallel matrix is stored row-wise; each processor owns the rows corresponding to its own partition. If a parallel matrix M is written as:

$$M = \begin{bmatrix} M_{cc} & M_{co} \\ M_{oc} & M_{oo} \end{bmatrix} = \begin{bmatrix} M_c \\ M_o \end{bmatrix}, \tag{3.1}$$

with the unknowns from the current partition ordered first, then the current processor owns M_c, or, M_{cc} and M_{co}. The matrices A, P and P^tAP are stored in this parallel data structure. All blocks of the globally coarse stiffness matrix P^tAP can be made accessible to all processors through global communication, which will not be expensive since the global coarse matrix is assumed to have only a few unknowns. Similarly, a parallel vector can be written as

$$v = \begin{bmatrix} v_c \\ v_o \end{bmatrix}; \tag{3.2}$$

the entries owned by the current processor are denoted by v_c. The coarse vector P^tv is made accessible to all processors as well.

For the simplicity of notation, we present the following algorithms under the assumption that the matrix A is symmetric. They can be generalized to the non-symmetric case with ease. The block structure of A_p is the same as that of M in equation 3.1. Since all the unknowns are stored on the pth processor, an additional subscription p will be used. Similarly, the block structure of r_p is also the same as that of v in equation 3.2, with an extra subscription p.

If π_p is given by equation 2.3, the blocks of the matrix A_p and the vector r_p are computed using:

```
ALGORITHM 3.1. (Setup up local matrix and residual for equation 2.3)
  1. $A_{p,cc} = A_{cc}$;
  2. $A_{p,co} = (AP)_c - A_{cc}P_{co}$;
  3. $A_{p,oc} = A^t_{p,co}$;
  4. $A_{p,oo} = (P^tAP)_{oo} - P^t_{co}(AP)_c + A^t_{p,do}P_{do}$;
  5. $r_{p,c} = r_c$;
  6. $r_{p,o} = (P^tr)_o - P^t_{co}r_c$.
```

If π_p is chosen as in equation 2.4, we compute A_p and r_p by computing their blocks individually.

```
ALGORITHM 3.2. (Setup up local matrix and residual for equation 2.4)
  1. $A_{p,cc} = A_{cc}$;
  2. $A_{p,co} = (A\tilde{P})_c - A_{cc}\tilde{P}_d$,
     where $\tilde{P}$ is the columns of $P$, which corresponds to the nonzero columns of $P_o$;
  3. $A_{p,oc} = A^t_{p,co}$;
  4. $A_{p,oo} = P^tAP - P^t_c(AP)_c - A_{poc}P_c$;
  5. $r_{p,c} = r_c$;
  6. $r_{p,o} = P^tr - P^t_cr_c$.
```

If π_p is given by equation 2.5, we notice that π_p can be obtained from its counterpart in equation 2.4 by adding up the columns corresponding to the duplicated interface unknowns. As a result, the local stiffness matrix and local residual can be obtained by designated row and column summations of the output of algorithm 3.2

```
ALGORITHM 3.3. (Setup up local matrix and residual for equation 2.5)
  1. Compute $A_p$ and $r_p$ using algorithm 3.2;
  2. if the $i$th row $A_{p,i}$ and the $j$th row $A_{p,j}$ correspond to the same unknown at
     the interface of the current partition, let $A_{p,i} = A_{p,i} + A_{p,j}$ and delete $A_{p,j}$;
  3. repeat step 2 for the columns of $A_p$;
  4. repeat step 2 for entries of $r_p$.
```

Assume the stiffness matrix has a fixed number of nonzeros per row. The memory complexity and the operator complexity of this algorithm is proportional to the number of unknowns per processor, which is bounded by the sum of the number of unknowns from the current partition and the number of coarse unknowns outside the current partition.

3.2. Three-level Algorithms. With a small coarsening level l, the two-level algorithm has good convergence rate in numerical experiments. However, as the number of processors grows, the local problem size increases significantly; solving local problems on each processor becomes more expensive. To reduce the expense of solving local problems, we introduced the three-level algorithm, where the local stiffness matrices consist of fine unknowns in the current partition, coarse unknowns at the level l in the neighboring partition. The unknowns from the remaining partitions are ignored. The three levels are fine, coarse and zero, as in figure 2.1.

For the three-level algorithms, we write a parallel matrix M in 3×3 blocks

$$M = \begin{bmatrix} M_{cc} & M_{cn} & M_{co} \\ M_{nc} & M_{nn} & M_{no} \\ M_{oc} & M_{on} & M_{oo} \end{bmatrix}.$$

The expanded matrix on processor p consists of fine unknowns in the current partition and neighboring partition.

$$M_{exp} = \begin{bmatrix} M_{cc} & M_{cn} \\ M_{nc} & M_{nn} \end{bmatrix}.$$

To generate matrix A_p at level l, we first build parallel matrices C_0 and C_l to collect expanded matrix A_{exp} and P_{exp} on each processor. Then we assemble a sequential matrix π_{exp} to coarsen A_{exp} in the neighboring partition.

At level l, C_l is generated according to how processor p is related to its neighboring processors. If processor p is connected to processors i and j, then we put identities to (s,p), $(s+1,i)$ and $(s+2,j)$ blocks of the pth partition of the parallel matrix C_l^t, where the row index s is given by an increment by one of the largest row index of partition $p-1$. The expanded stiffness matrix and the expanded interpolation matrix are given by

$$\begin{aligned} A_{p,exp} &= (C_0^t A C_0)_{cc}, \\ P_{p,exp} &= (C_0^t P_l C_l)_{cc}. \end{aligned} \tag{3.3}$$

Whether processor p is connected to processor i is determined by examining the level-l global interpolation matrix P_l. If the (p,i) block of P_l is nonzero, then p is connected to i. This relationship is not necessary symmetric, e.g., p is connected to i does not indicate that i is connected to p.

A detailed example for computing the matrix C^t is described here. Processor 1 is connected to processors 2 and 3; processors 2, 3 and 4 are separated from each other, see figure 3.1. The matrix C^t has four partitions. Identities are given to $(1,1)$, $(2,2)$, $(3,3)$ blocks of the first partition. $(4,2)$, $(5,3)$ and $(6,4)$ blocks are given to partitions 2, 3 and 4 respectively.

$$C^t = \begin{bmatrix} I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \\ -- & -- & -- & -- \\ 0 & I & 0 & 0 \\ -- & -- & -- & -- \\ 0 & 0 & I & 0 \\ -- & -- & -- & -- \\ 0 & 0 & 0 & I \end{bmatrix}.$$

Since all the interpolation information in the expanded partitions is available locally, in the matrix $P_{p,exp}$, it is easy to generate the local expanded interpolation matrix $\pi_{p,exp}$. We compute sequential matrix π_{exp} according to equation 2.6 or 2.7. No communication is require at this step. The algorithm for building A_p and r_p is then given below:

Algorithm 3.4. *(Setup up local matrix and residual for equation 2.3 or 2.5)*

```
1. At level $l$, generate parallel matrices $C_0^t$ and $C_l^t$;
2. compute $A_{exp}$ and $P_{exp}$;
3. generate $\pi_{exp}$;
4. compute $A_p = \pi_{exp}^t A_{exp} \pi_{exp}$ using equations 3.3;
5. compute $r_p = \pi_{exp}^t (C_0^t r)_c$.
```

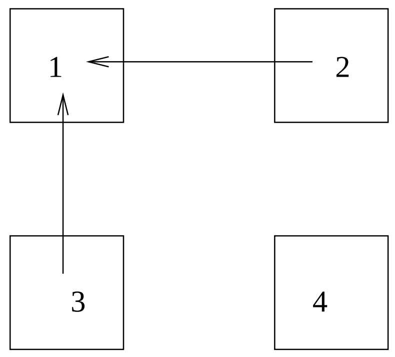

FIG. 3.1. *Connected and isolated processors*

As in section 3.1, the memory and computation complexity is proportional to the number of unknowns per processor. In the three-level algorithms, the number of unknowns per processor is the sum of the number of unknowns in the current partition and the number of coarse unknowns from the neighboring partitions. The operator complexity is proportional to the number of unknowns per processor. The communication complexity is proportional to the number of coarse unknowns outside the current partition on each processor.

The complexity of the sequential AMG algorithms is usually measured with two indicators: the rate of convergence and the operator complexity. The first factor is equivalent to the iteration count of an algorithm; the second one is equivalent to the computing time per iteration; and their product gives the total computing cost of the solving stage. However, to measure the complexity of a parallel algorithm, we need to introduce a third factor, which is the communication complexity. Together with the operator complexity, the communication complexity decides the computing time required for each iteration. Our parallel AMG algorithm aims at reducing the total solving time by doing the majority of communication in the initialization stage. In the solving stage, the communication complexity is reduced to be comparable to that of a simple iterative method.

3.3. FocusDD Library. This parallel domain decomposition solver/preconditioner is implemented using object oriented programming in the C language. A library called FocusDD is built on the HYPRE library. The interface functions are

```
solver = hypre_FocusDDCreate();
hypre_FocusDDSetup( solver );
hypre_FocusDDSolve( solver, A, f, u);
hypre_FocusDDDestroy( solver );
```

Because this solver is not symmetric, the GMRES method rather than the CG method is used to accelerate convergence.

4. Numerical Results.

The 3D discrete Poisson equation is solved with our proposed algorithms. A scalability study is conducted with varying number of unknowns and processors. The number of unknowns per processor ranges from $1k$ to $10k$; the number of processors ranges from 2 to 128. The domain of the PDE is partitioned into cubic subdomains to test scalability. The iteration counts and computing time needed to obtain a residual norm reduction of 10^{-6} is compared with that of BommerAMG. The tests were run on a Linux-based Beowulf cluster, consisting of 16 dual 100MB CISCO 2950G Ethernet switch. This cluster runs NAPACI rocks version of Linux (based on RedHat7.1), and employs the MPICH implementation of message passing interface (MPI).

TABLE 4.1
FocusDD solver, type 0

np	npp	iter	γ	setup time	solve time
2	1000	11	0.47	0.18	0.05
4	1000	12	0.25	0.27	0.07
8	1000	17	0.43	0.96	0.23
16	1000	18	0.40	2.37	0.40
32	1000	17	0.39	5.46	0.67
64	1000	22	0.43	49.55	3.62

TABLE 4.2
FocusDD solver, type 1

np	npp	iter	γ	setup time	solve time
2	1000	9	0.20	0.22	0.05
4	1000	10	0.16	0.32	0.06
8	1000	14	0.39	1.53	0.20
16	1000	15	0.35	2.63	0.36
32	1000	15	0.35	6.29	0.78
64	1000	17	0.43	53.08	2.97
128	1000	17	0.45	355.17	7.47

TABLE 4.3
FocusDD solver, type 2

np	npp	iter	γ	setup time	solve time
2	1000	13	0.25	0.26	0.07
4	1000	13	0.33	0.33	0.10
8	1000	22	0.48	1.60	0.31
16	1000	24	0.66	2.52	0.55
32	1000	23	0.40	37.48	1.19
64	1000	29	0.63	44.61	4.62

The convergence behavior of the algorithms implemented in FocusDD is shown in tables 4.2–4.5. The convergence result of BoomerAMG is shown in table 4.6, as a reference. In the header, *np*, stands for the number of processors; *npp* stands for the number of unknowns per processor; *iter* stands for the number of iterations required to reduce the residual norm by a factor of 10^{-6}; γ stands for the approximate convergence factor; *setup time* and *solve time* give the time used in the initialization stage and the solving stage, respectively, in seconds.

Among the FocusDD solvers, solver types 0–2 are two-level algorithms. Type 0 uses the local interpolation matrix π_p in equation 2.3; type 1 uses π_p in equation 2.4;

TABLE 4.4
FocusDD solver, type 3

np	npp	iter	γ	setup time	solve time
2	1000	1	0.00	0.17	0.01
4	1000	3	0.00	0.36	0.05
8	1000	7	0.39	0.86	0.17
16	1000	13	0.20	2.61	0.33
32	1000	20	0.55	5.04	0.55
64	1000	38	0.78	22.68	1.61
128	1000	80	0.81	67.29	3.70

TABLE 4.5
FocusDD solver, type 4

np	npp	iter	γ	setup time	solve time
2	1000	9	0.20	0.20	0.05
4	1000	10	0.16	0.33	0.08
8	1000	14	0.39	1.07	0.21
16	1000	15	0.35	2.60	0.35
32	1000	15	0.35	7.83	0.69
64	1000	17	0.43	55.51	3.09

type 2 uses π_p in equation 2.5. Types 3 and 5 are three-level algorithms. In both types, π_p is chosen as in equation 2.7. The difference between these two types is the following: type-3 solver extends the local stiffness matrix to the whole neighboring partitions; type-5 solver only extends the local stiffness matrix to a few layers into the neighboring partitions. Type-4 solver uses the overlapping Schwartz method described in section 2.4. In every iteration, it uses a two-level type-1 solver, followed by a three-level type-3 solver.

In the two-level algorithms, as $npp = 1000$, the iteration count grows slowly as np increases from 2 to 128, see tables 4.2–4.3. However, the solving time per iteration grows fast, since the size of the global coarse problem increases together with np. Therefore, the total solving time increases regardless of the iteration counts. The results of BoomerAMG solver is shown in table 4.6. In comparison, BoomerAMG uses less setup time, but more solve time than FocusDD. The result of a three-level algorithm is shown in table 4.4. Its solving time is about half of that of BoomerAMG, when $np = 4, 8, 16$. The difference becomes larger when np is greater than 16.

An additive Schwartz algorithm is also tested, results shown in table 4.5. In each iteration, this algorithm includes a three-level solver followed by a two-level solver. These two solvers are accelerated by the GMRES method. $level = 1$ is chosen for the neighboring partition, in the three-level solver. The coarsest level is chosen for the two-level solver. The iteration counts is almost stable; yet computing time per iteration increases slightly as np increases. Among the five algorithms in FocusDD, the three-level algorithm in table 4.4 is the most efficient in solving time.

The above results are summaries in figures 4.1 and 4.2. As can be seen from figure 4.1, BoomerAMG has a constant iteration count 1, for $np = 2, \cdots, 64$. In FocusDD, both the two-level algorithms and the overlapping Schwartz method have an almost constant iteration count. The iteration count of the three-level algorithm grows with np. However, in terms of solving time, the three-level algorithm is the

TABLE 4.6
BoomerAMG solver

np	npp	iter	γ	setup time	solve time
2	1000	1	0.00	0.05	0.01
4	1000	1	0.00	0.11	0.14
8	1000	1	0.00	0.42	0.31
16	1000	1	0.00	1.44	0.67
32	1000	1	0.00	3.32	1.41
64	1000	1	0.00	18.17	5.73
128	1000	1	0.00	59.70	20.57

most efficient, see figure 4.2. All five types of solvers implemented in FocusDD require less solving time than BoomerAMG.

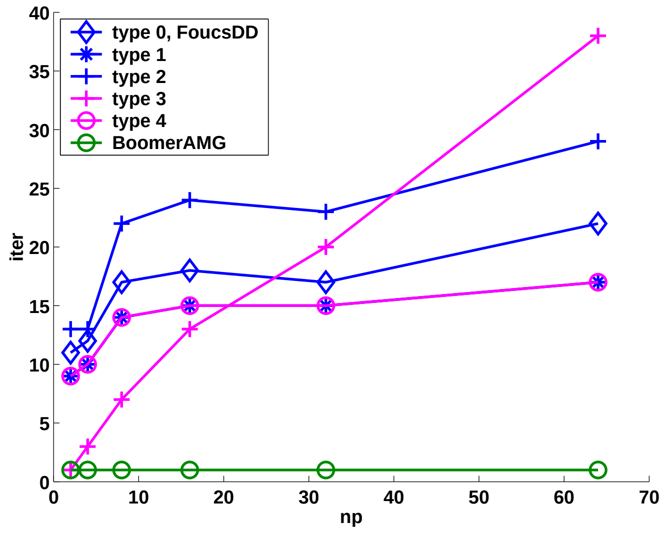

FIG. 4.1. *Iteration counts, npp* = 1000.

As *npp* = 10000, it is not appropriate to use the three-level algorithm which extends the local problem to one coarser level of all the neighboring partitions, since the communication cost will be quite daunting. Instead, we restrict the extension to only a few layers outside the current partition. In our test problem, it is 4 layers. Table 4.7 shows the results of BoomerAMG, as a reference. Table 4.8 shows the results of FocusDD, using the three-level algorithm with 4 expanding layers. FocusDD uses less time per iteration, which indicates that the stepwise computing and communication complexity are less than that of BoomerAMG. However, since the iteration counts of FocusDD is much bigger. BoomerAMG seems to be more efficient than FocusDD in both setup time and solving time.

Based on the above observation, our proposed algorithm is indeed effective in reducing stepwise communication complexity during solving stage. However, the other two measures of parallel efficiency, computing complexity and convergence rate, may suffer. Another problem is that the initialization stage is quite expensive. Future work will be directed to improving these aspects. For example, it is possible to de-

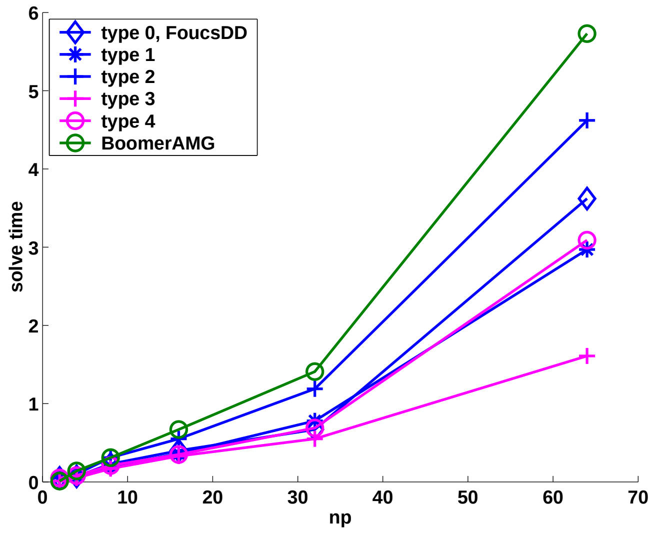

FIG. 4.2. *Solving time, npp* $= 1000$.

TABLE 4.7
BoomerAMG solver

np	npp	iter	γ	setup time	solve time
2	10000	1	0.00	0.85	0.79
4	10000	1	0.00	1.42	1.03
8	10000	1	0.00	5.70	2.09
16	10000	1	0.00	15.48	3.47
32	10000	1	0.00	29.17	5.10
64	10000	1	0.00	162.73	16.07

sign an algorithm which generates an hierarchy of extended stiffness matrices and interpolation matrices directly from the results of BoomerAMG, eliminating the need of re-initialization by the multigraph solver. Furthermore, since the complete information about the hierarchy of stiffness matrices is retained, this new design has a potential to be as efficient as the domain decomposition GMG solvers presented in [6, 20].

REFERENCES

[1] HYPRE library. http://www.llnl.gov/CASC/hypre/.

[2] R. E. Bank and M. Holst. A new paradigm for parallel adaptive meshing algorithms. *SIAM J. on Scientific Computing*, 22:1411–1443, 2000.

[3] R. E. Bank and M. Holst. A new paradigm for parallel adaptive meshing algorithms. *SIAM Rev.*, 45(2), 2003.

[4] R. E. Bank and P. K. Jimack. A new parallel domain decomposition method for the adaptive finite element solution of elliptic partial differential equations. *Concurrency*, to appear.

[5] R. E. Bank, P. K. Jimack, S. Nadeem, and N. S.V. A weakly overlapping domain decomposition preconditioner for the finite element solution of elliptic partial differential equations. *SIAM J. on Scientific Computing*, to appear.

[6] R. E. Bank and S. Lu. A domain decomposition solver for parallel adaptive meshing paradigm. *SIAM J. Sci. Comput.*, 26:105–127, 2004.

TABLE 4.8
FocusDD solver, type 5.

np	npp	iter	γ	setup time	solve time
2	10000	4	0.02	13.03	0.78
4	10000	5	0.02	17.57	1.17
8	10000	17	0.28	37.47	5.91
16	10000	20	0.37	53.35	8.28
32	10000	22	0.53	71.84	9.80
64	10000	42	0.78	229.67	26.64

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

[8] A. Brandt. Algebraic multigrid theory: The symmetric case. In *Preliminary porceedings for the international multigrid conference*, Copper Mountain, Colorado, 1983.

[9] A. Brandt. Algebraic multigrid (AMG) for sparse matrix equations. In D. Evans, editor, *Sparsity and its applications (Loughborough, 1983)*, pages 257–284. Cambridge University Press, Cambridge, 1985.

[10] A. Brandt. Algebraic multigrid theory: The symmetric case. *Appl. Math. Comput.*, 19:23–56, 1986.

[11] A. Brandt, S. F. McCormick, and J. W. Ruge. Algebraic multigrid (AMG) for automatic multigrid solutions with applications to geodetic computations. Report, 1982.

[12] M. Brezina, A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, and J. W. Ruge. Algebraic multigrid based on element interpolation(AMGe). *SIAM J. Sci. Comput.*, 22:1570–1592, 2000.

[13] Q. Chang and Z. Huang. Efficient algebraic multigrid algorithms and their convergence. *SIAM J. Sci. Comput.*, 24(2):597–618 (electronic), 2002.

[14] Q. Chang, Y. S. Wong, and H. Fu. On the algebraic multigrid method. *J. Comput. Phys.*, 125(2):279–292, 1996.

[15] A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, G. N. Miranda, and J. W. Ruge. Robustness and scalability of algebraic multigrid. *SIAM J. Sci. Comput.*, 21(5):1886–1908 (electronic), 2000. Iterative methods for solving systems of algebraic equations(Copper Mountain, CO, 1998).

[16] G. Haase, M. Kuhn, and S. Reitzinger. parallel algebraic multigrid methods on distributed memory computers. *SIAM J. Sci. Comput.*, 24(2):410–427, 2002.

[17] V. E. Henson and U. M. Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41:155–177, 2002.

[18] R. Kimmel and I. Yavneb. An algebraic ultigrid approach for image analysis. *SIAM J. Sci. Comput.*, 24(4):1218–1231, 2003.

[19] A. Krechel and Stüben. Parallel algebraic multigrid based on subdomain blocking. *Parallel Comput.*, 27:1009–1031, 2001.

[20] S. Lu. *Scalable parallel multilevel algorithms for solving partial differential equations.* PhD thesis, University of California, San Diego, 2004.

[21] S. Meynen, A. Boersma, and P. Wriggers. Application of a parallel algebraic multigrid method for the solution of elastoplastic shell problems. *Numer. Linear Algebra Appl.*, 4(3):223–238, 1997.

[22] M. Raw. A coupled algebraic multigrid method for the 3D Navier-Stokes equations. In *Fast solvers for flow problems (Kiel, 1994)*, volume 49 of *Notes Numer. Fluid Mech.*, pages 204–215. Vieweg, Braunschweig, 1995.

[23] S. Reitzinger and J. Schöberl. An algebraic multigrid method for finite element discretizations with edge elements. *Numer. Linear Algebra Appl.*, 9(3):223–238, 2002.

[24] J. W. Ruge and Stüben. Efficient solution of finite difference and finite element equations by algebraic multigrid (AMG). In D. J. Paddon and H. Holstein, editors, *Multigrid methods for integral and differential equations, The Institute of Mathematics and its Applications Conference Series*, pages 169–212. Clarendon Press, Oxford, 1985.

[25] J. W. Ruge and Stüben. Algebraic multigrid. In S. McCormick, editor, *Multigrid methods*, pages 73–130. Philandelphia, PA, 1987.

[26] K. Stüben. Algebraic multigrid (AMG): experiences and comparisons. *Appl. Math. Comput.*, 13:419–452, 1983.

[27] K. Stüben. A review of algebraic multigrid. *J. Comput. Appl. Math.*, 128(1-2):281–309, 2001. Numerical analysis 2000, Vol. VII, Partial differential equations.

[28] K. Stüben, U. Trottenberg, and K. Witsch. Software development based on multigrid techniques. In B. Enquist and T. Smedsaas, editors, *Proceedings IFIP-Conference on PDE software, modules, interfaces and systems*, Söderköping, Sweden, 1983.

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

[30] C. Wagner. On the algebraic construction of multilevel transfer operators. *Computing*, 65:73–95, 2000.