Non-standard parallel solution strategies for distributed sparse linear systems*

Yousef Saad[1] and Maria Sosonkina[2]

[1] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MM 55455, USA,
saad@cs.umn.edu,
WWW home page: http://www.cs.umn.edu/ saad

[2] Department of Computer Science, University of Minnesota, Duluth, 320 Heller Hall, 10 University Drive, Duluth, Minnesota 55812-2496. masha@d.umn.edu.

Abstract. A number of techniques are described for solving sparse linear systems on parallel platforms. The general approach used is a domain-decomposition type method in which a processor is assigned a certain number of rows of the linear system to be solved. Strategies that are discussed include non-standard graph partitioners, and a forced load-balance technique for the local iterations. A common practice when partitioning a graph is to seek to minimize the number of cut-edges and to have an equal number of equations per processor. It is shown that partitioners that take into account the values of the matrix entries may be more effective.

1 Introduction

Recent years have seen a maturation of parallel processing to a point where the methodology is now beginning to enter many engineering and scientific applications. The innermost part of these applications often requires the solution of large sparse linear systems of equations. The most common architecture used is that of a distributed memory computer, using MPI for message passing. The most natural process for solving Partial Differential Equations and sparse linear systems on distributed memory computers is to employ strategies based on domain decomposition. A typical finite element simulation for example, requires the following steps: (1) The physical mesh is generated, typically on one processor; (2) The mesh is partitioned using a number of publically available tools; (3) The element matrices and right-hand sides are generated in each processor independently; (4) Finally, a solution process, typically based on iterative methods, for the resulting distributed system is undertaken. This course of action seems natural and straightforward. It comes, however, with a few challenging questions.

The first of them is related to partitioning. What partitioning approaches will lead to the best overall performance of the solver? Most current partitioners

* Work supported by NSF under grant CCR-9618827, and in part by the Minnesota Supercomputer Institute.

will simply divide up the graph aiming at obtaining about the same number of points in each processor and at reducing the number of edge cuts. A number of heuristics have been developed with this strategy in mind, see e.g., [10, 7, 13]. However, it is easy to imagine that this is far from perfect. First, if load balancing is the main criterion (ignoring communication for a moment) then clearly, the number of points assigned to each processor is not a good measure. One can imagine, for example, a technique based on attempting to equalize the time spent on matrix-vector products in which case, an algorithm that would distribute edges rather than vertices equally would be more appropriate. These two strategies may lead to a similar distribution in many cases, but not always. Another rather complex issue is that the partitioning can affect the quality of the preconditioning in a way that is hard to predict. We may obtain a partitioning that has perfect load balance and a minimal number of cut-edges but which may lead to an unacceptable increase in the number of iterations. This happens almost systematically when the matrix arises from highly discontinuous problems.

This paper illustrates these issues by devising a number of alternative strategies that can be used for partitioning a graph and reducing idle time during an iteration. The intent is not to develop a new general method but rather to show that a few alternative approaches can lead to effective solution methods.

2 Graph Partitioning Concepts

The very first task that a programmer faces when solving a problem on a distributed memory parallel computer, be it a dense or a sparse linear system is to decide how to map the data into the processors. We call a *map* of V, any set $V_1, V_2, \ldots, V_s$, of subsets of the vertex set V, whose union is equal to V:

$$V_i \subseteq V \ , \quad \bigcup_{i=1,s} V_i = V.$$

When all the subsets V_i are not pairwise disjoint, the term partition conflicts with common usage, but we will use the term overlapping partition in this case. The most general way of describing a node-to-processor mapping is to set up a list, containing all the nodes that are mapped to each processor. Thus, in the example shown in Figure 1, the list $\{1, 2, 5, 6\}$ is assigned to Processor 1, the list $\{3, 4\}$ is assigned to Processor 2, the list $\{7, 8, 11, 12\}$ is assigned to Processor 3, and the list $\{9, 10\}$ is assigned to Processor 4. Another representation which is sometimes useful is to set-up an array which lists for each node the processor to which it belongs. This is important when setting up the local data structure in the preprocessing phase of domain decomposition type algorithms [16].

There has been a flurry of activity in finding good partitionings of graphs. Among the most popular techniques are the spectral bisection method [13] and Metis [10]. These methods attempt to provide a partition that will have good load balancing and a small number of edge cuts. A few algorithms to accomplish this have been described in [1, 6, 13, 12, 4, 7, 10].

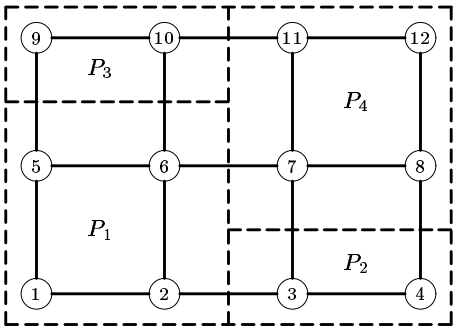

Fig. 1. Mapping of a simple 4×3 mesh to 4 processors.

2.1 A few graph-based partitioners

The Breadth-First-Search (BFS) algorithm is at the basis of many partitioning techniques, including those to be described in this paper. BFS is essentially a level-by-level traversal of a graph. It starts with a given vertex v_0 which will constitute the level 0. Then all nodes adjacent to v_0 will be visited – and these constitute level 1. In the third step, all non-visited vertices that are adjacent to vertices belonging to level 1, will be visited and they will constitute Level 2, etc.

We note that BFS does need not start with a single node. We can, for example, start with a known level, i.e., a set of nodes forming a string of connected nodes. The one-way partitioning algorithm simply traverse the nodes in the Breadth-First-Search order and assigns nodes to partitions until a given number of assigned nodes is reached.

ALGORITHM **21** *One-way-partitioning using level-sets*

```
1.  Input: nlev, levels, ip (number of nodes per partition)
2.  Output: ndom, node-to-processor list.
3.  Start: ndom = 1; siz = 0;
4.  Loop: For lev =1, nlev Do
5.          For each j in levels(lev) Do
6.              add j to dom(ndom); siz = siz+1;
7.              If (siz .ge. ip) then
8.                  ndom = ndom+1; siz = 0;
9.              EndIf
10.         EndDo
```

In an actual implementation, some care must be exercised to obtain equal sized domains, and to avoid having a last subdomain consisting of the points

that are left-over in the very last step. For example, if $n = 101$ and $ip = 10$, we will have 10 subdomains of size 10 each and one of size one. This can be prevented by adjusting the ip parameter as we proceed. Two-way partitioning consists of two applications of one-way partitionings. In a first pass we determine a one-way partition. Then each of the subdomains created is again partitioned using the one-way partitioning algorithm. This strategy will be referred to as the "double-striping" algorithm. It uses 2 parameters: p_1 the number of partitions in the first one-way partitioning, and p_2 the number of sub-partitions in each of the p_1 partitions resulting from the first pass.

A critical ingredient in the efficiency of the procedure is the starting node. A well-known procedure for this is a heuristic for finding a so-called pseudo-peripheral node [5]. This procedure starts with an arbitrary node x and performs a BFS from this node. It then records the node y that is farthest away from x and the corresponding distance $dist(x, y)$. We then assign $x := y$ and perform a new traversal from x. We repeat the process and stop when $dist(x, y)$ does not vary between two successive traversals.

2.2 Level-set expansion algorithms

As was noted earlier, the Breadth First Search traversal can start from several independent vertices at once instead of one node only. Level-set expansion algorithms consist of building the subdomains by a using a BFS traversal from a number of centers. Thus, these algorithms consist of two phases. The first phase finds 'center' nodes for each partition from which to expand each subdomain. Ideally, these centers are vertices that are far apart from one another, in a graph theory sense. In the second phase these points are used to generate the subdomains using a BFS expansion from them.

Next, we present a version of the algorithm that is sequential with respect to the domains but it is clear that the process is inherently parallel.

ALGORITHM **22** *Level-set-expansion*

1. **Start:**
2. *Find an initial set of 'coarse mesh' vertices 3.* $v_1, \ldots, v_{ndom}$
4. *For* $i = 1, 2, \ldots, ndom$ *Do* $label(v_i) := i$.
5. *Define* $levset := \{v_1, \ldots, v_{ndom}\}$ *and* $nodes = ndom$
6. **2. Loop:** *While (nodes <n) Do*
7. $Next\_levset = \phi$
8. *For each* v_j *in* $levset$ *Do*
9. *for each neighbor* v_k *of* v_j *s.t.* $label(v_k) = 0$ *Do*
10. $Next\_levset := Next\_levset \bigcup \{v_k\}$
11. $label(v_k) := label(v_j)$
12. $nodes = nodes + 1$
13. *EndDo*
14. *EndDo*
15. $levset := Next\_levset$

16. EndWhile

The algorithm starts with one node in each processor then expands by adding level sets until all points are labeled. The indicator that is assigned on each node to indicate whether or not the note has already been visited is now a label and the vertices will inherit the labels of their parents in the traversal process. At the end, all nodes having the same label will constitute a subdomain. We must assume here that the initial graph is connected or that there is at least one starting node in each connected component. The parallel version of this algorithm consists of assigning each starting node to a different processor, then expanding the level sets independently. At some point there will be conflicts, i.e., two processors will attempt to 'acquire' the same node which belongs to two level sets originating from two different starting nodes. In such cases, the host program must arbitrate. Our current implementation uses a first-come first served rule, but there are several possible improvements which are not considered here.

The next question is how to find the center points. There are at least three possible options. First, if a coarse mesh is already available from the discretization then the nodes of this mesh can be taken as the centers. If a coarse mesh is not available we can alternatively use points provided from a two-way partitioning algorithm. Second, if the coordinates of the nodes are available then one can easily select the centers from simple geometrical considerations. For example, for a rectangular 2-D mesh, we can choose points that are uniformly distributed in each direction. Other alternatives are required for the cases where only the graph is known and coordinate information is not available.

In general, the two-way partitioning algorithm does not provide as good a splitting of the graph as some of the well-known alternatives such as the Recursive Spectral Bisection technique [13]. It is, however, rather inexpensive to obtain. As a result, we can use this partitioning only to get the centers for the Level-Set Expansion. For example, we can simply take the middle node in the subdomain as a center. The resulting partitioning will be far better than the original two-way partitioning in general. An illustration of the process is given in Figure 2.

2.3 Partitioning strategies using shortest path methods

Partitioning is a form of reordering and as such it may have an important effect on the quality of partitioning. The purpose of this section is only to illustrate this point and propose, not a solution, but some general guidelines toward developing effective partitioners for iterative solvers.

The key point is to take into account the matrix values when partitioning the problem. This can be done using weights on edges. The weight values are based on the absolute magnitudes of the matrix entries associated with the edges. In particular, if e_{ij} is the edge connecting vertices i and j, then the weight $w(e_{ij})$ of this edge is $1/(1 + |a_{ij}| + |a_{ji}|)$, where both matrix entries a_{ij} and a_{ji} are

associated with edge e_{ij} under the assumption that the matrix is structurally symmetric. The hypothesis used here is that keeping together those nodes which are strongly coupled would lead to a better preconditioning. Weights have been used in partitioning methods, such as those developed in public domain codes. A drawback of such partitionings, however, is that the decision with respect to the weights is made only locally, that is, the weights are considered only to choose among the possible edge-cut candidates.

To achieve a better quality preconditioning, we would like to "propagate" the weight-related information and group together several matrix entries whose added weights are small. The shortest path algorithm may be employed for this purpose. This algorithm finds the path with the smallest weight from a given node to all the nodes in a graph. Note that this shortest path algorithm starts with a single node. Finding the shortest path results in recording one by one the nodes constituting this path in a proper order, which we will call a shortest-path ordering. Then the nodes are gathered in the subdomains based on this shortest-path ordering rather than on a level-set ordering. At the same time, we would like to keep the edge cut small, i.e., we do not let any shortest path have a large number of (cheap) links. In other words, we would like to preserve the quality of partitions produced by a level-set expansion from the chosen center points. Thus, we introduce a weight coefficient indicating the relative location of a node in the level-set structure produced by BFS starting from a center point. For each edge e_{ij}, this coefficient may be computed, for example, as the sum of the levels at which the incident with this edge nodes are located. In general, the partitioning algorithm consists of the following major steps:

1. Find domain centers.
2. Do level-set expansion from these centers.
3. Record the level for each node.
4. Compute the weight coefficients.
5. Multiply weights by the coefficients.
6. Do shortest path ordering of the nodes.
7. Collect nodes into subdomains.

The full shortest-path algorithm is quite expensive: A standard implementation costs $\mathcal{O}((|V| + |E|) \log |V|)$. Thus, some heuristic approximations were used instead of the complete shortest path algorithm. A test has been performed on the RAEFSKY3 matrix [2]. This matrix is of size 21,200 and has 1,488,768 of nonzeros. The Schur-LU preconditioner [17] has been applied to solve the corresponding linear system using FGMRES(20) [15] on 16 processors. A brief description of the Schur-LU preconditioner is given in Section 3. Table 1 shows a comparison between using a standard level-set expansion versus a method using a full shortest-path ordering in the expansion. The times are in seconds on the Paragon parallel computer. Note that these timings measure the solution phase only. The major time gains occur in the preconditioning phase — 142.04 vs. 97.84 seconds — which is not surprising since the preconditioning application, being expensive, benefits the most from improving the quality of the local ma-

trices. The importance of using the knowledge of matrix values in partitioning can be demonstrated in another example discussed next.

	Time	Iterations
Standard	175.27	161
Shortest-Path	122.95	106

Table 1. Impact of two different partitioning strategies on the performance of the Schur-LU preconditioner.

2.4 Partitioning for problems with discontinuities

Consider an elliptic partial differential equation of the form

$$-\frac{\partial}{\partial x}\left(a\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(b\frac{\partial u}{\partial y}\right)=h \tag{1}$$

on rectangular regions with general mixed-type boundary conditions. In the test problems, the regions are the square $\Omega=(0,1)^2$, or the cube $\Omega=(0,1)^3$; the Dirichlet condition $u=0$ is always used on the boundary. Only the discretized matrix is of importance, since the right-hand side will be created artificially. Therefore, the right-hand side h is not relevant here. The mesh is 96×96 grid points *per processor*, meaning that the problem size is scaled with the number of processors: the more processor used, the larger the problem solved. For example, on 4 processors, the problem size is $96\times96\times4=36,864$; on 16 processors, the problem size is $96\times96\times16=147,456$.

In the region $0.25<x,y<0.75$, the coefficient $a=100$ (Figure 3) and $a=1$ elsewhere, while the function b is constant and equal to 1. We can partition the resulting mesh without consideration given to the coefficients or we can partition it by trying to ensure that the discontinuity lines will not cross subdomains. Having coefficients $a=100$ and $b=1$ in the region creates the discontinuity in only one direction, which already shows the advantages of treating this region separately and allows a greater freedom in partitioning it separately. We use a two-way level-set partitioning ("double-stripe") as described in Section 2.1. This consists of taking a pseudo-peripheral node [5], then doing a breadth-first traversal from the peripheral node and keeping as many levels in the traversal as needed to have about n/p_1 nodes in each processor. The process is repeated to partition each of the resulting subgraphs into p_2 subpartitions. The result is a partition into $p=p_1\times p_2$ subgraphs, where p must be multiple of 4 in our example.

Table 2 shows the solution times using this algorithm (`DoubleStripe`) and its modification (`DoubleStripe_m`) in which the partitioning is done in two phases. First, the double-stripe algorithm is applied to the area outside the region ("low

coefficient a" area) only to partition it into $3p/4$ processors (Figure 4—left). Second, the rest of the processors ($p/4$) is assigned to the "high coefficient a" area inside the region using the same algorithm.

	Time			Iterations		
PEs	16	24	48	16	24	48
`DoubleStripe`	9.66	14.34	51.29	81	132	341
`DoubleStripe_m`	5.58	6.33	6.17	50	61	58
`ByHand`	6.39	12.73	9.98	52	120	87
`ByHand_m`	6.3	7.34	7.25	53	62	59

Table 2. Solution times (Paragon seconds), numbers of outer iterations for the Schur-LU preconditioner with four different partitioning strategies.

One might argue that using a general purpose partitioning for this problem is not adequate since we can do the partitioning "By hand", i.e., provide a rectangular partitioning for this regular mesh. In Table 2, `ByHand` represents such a partitioning. To consider separately the "low coefficient a" region, we proceed in the same way as with double-stripe partitioning and assign the region to the $p/4$ processors as shown in Figure 4(right). The results for the two-phase partitioning "By hand" are labeled `ByHand_m` in Table 2.

3 Solution of Distributed Sparse Systems

A distributed sparse linear system is a collection of sets of equations that are assigned to different processors. Each equation of the global linear system must be assigned to at least one processor. When equation number i is assigned to processor p, it is always assumed that the corresponding variable i is also assigned to processor p. A pair consisting of an equation and the corresponding unknown is sometimes referred to as a node. Overlapping refers to situations in which a given node is assigned to more that one processor. The parallel solution of a sparse linear system begins with partitioning the adjacency graph of the coefficient matrix. The linear system is then distributed by assigning the equations to processors according to the partitioning. When this is done, three types of unknowns can be distinguished: (1) Interior unknowns that are coupled only with local equations; (2) Local interface unknowns that are coupled with both non-local (external) and local equations; and (3) External interface unknowns that belong to other subdomains and are coupled with local equations. This setting which is illustrated in Figure 5, is common to most packages for parallel iterative solution methods [14, 16, 8, 19, 19, 3, 18, 9].

The matrix assigned to a certain processor is split into two parts: the *local* matrix A_i, which acts on the local variables and an *interface matrix* X_i, which acts on the external variables. Accordingly, the local equations can be written

as follows:

$$A_i x_i + X_i y_{i,ext} = b_i. \tag{2}$$

where x_i represents the vector of local unknowns, $y_{i,ext}$ are the external interface variables, and b_i is the local part of the right-hand side vector. It is common to reorder the local equations in such a way that the interface points are listed last after the interior points. This ordering leads to an improved interprocessor communication and to reduced local indirect addressing during matrix-vector multiplication. Thus, the local variables form a local vector of unknowns x_i which is split into two parts: the subvector u_i of internal vector components followed by the subvector y_i of local interface vector components. The right-hand side b_i is conformally split into the subvectors f_i and g_i. When the block is partitioned according to this splitting, the local equations (2) can be written as follows:

$$\begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix}. \tag{3}$$

Here, N_i is the set of indices for subdomains that are neighbors to the subdomain i. The term $E_{ij} y_j$ is a part of the product $X_i y_{i,ext}$ which reflects the contribution to the local equation from the neighboring subdomain j. The result of the multiplication by X_i affects only the local interface unknowns, which is indicated by a zero in the top part of the second term of the left-hand side of (3).

3.1 Distributed Krylov subspace solvers

Implementation issues when developing Preconditioned Krylov subspace algorithms for solving distributed sparse systems have been discussed elsewhere, see e.g., [16, 8, 19, 3, 15]. Once the data structure associated with the local data has been built in each processor, then the operations required for implementing a Krylov subspace method are (1) vector operations such as SAXPY and dot products, (2) matrix vector products, and (3) preconditioning operations. The SAXPY operations are entirely local, since vectors are all partitioned conformally. The dot products require a global sum of partial inner products. This reduction operation requires communication but if the domains are large enough, the related overhead is usually small.

To multiply a given vector x by the global matrix, we only need to multiply the local matrix by the local part of x to obtain a vector z then get the external interface variables, multiply them by the X matrix and add the result to z. Thus, there are three costs to be added: the cost of the local matvec, then the cost of the exchange of interface data and finally the cost of the second matvec.

The most important operation when solving distributed sparse linear systems is undoubtedly the preconditioning operation. A discussion of preconditioners is beyond the scope of this paper. Here we only discuss two such methods for the sake of completeness.

The simplest domain-decomposition preconditioner is the additive Schwarz procedure, which is a form of the block Jacobi iteration, where the blocks refer to matrices associated with entire subdomains.

ALGORITHM **31** *Additive Schwarz*

1. *Obtain external data* $y_{i,ext}$
2. *Compute (update) local residual* $r_i = (b - Ax)_i = b_i - A_i x_i - X_i y_{i,ext}$
3. *Solve* $A_i \delta_i = r_i$
4. *Update solution* $x_i = x_i + \delta_i$

The systems which arise in line 3, are solved by either a standard (sequential) ILUT preconditioner [15] combined with GMRES or the application of one ILU preconditioning operation. Of particular interest in this context are the overlapping additive Schwarz methods. In the domain decomposition literature [18] it is known that overlapping is a good strategy to reduce the number of steps.

Another preconditioning method described in [17] is the Schur-LU preconditioner. The main step in this technique is to solve approximately the Schur complement system, i.e., the (global) system which involves the interface variables y. This system is obtained by eliminating the variable u_i from equation (3), using the first equation. This global system in y can be solved approximately by a form of block Jacobi preconditioner. An ILUT factorization for the matrix A_i yields as a by-product an ILUT factorization for the local Schur complement. This is used to precondition the global Schur system. Once the approximation to the y variable is obtained, the u variables are extracted. The step of obtaining the approximate u_i, y_i pair in this manner from a right-hand side constitutes one step of the Schur-LU preconditioner. For further details, see [17].

3.2 Reducing idle time in preconditioning operations

For the local preconditioning strategies, such as Additive Schwarz, the amount of work each processor accomplishes in the preconditioning application is different and depends on the properties of the local submatrices. Since the properties of the local submatrices may vary greatly, the times of the preconditioning phase may also differ substantially leading to a load imbalance among processors. Thus when the processor synchronizations take place (in the orthogonalization phase of FGMRES and during the matrix-vector product computation), the processors with a small preconditioning workload must wait for the rest of the processors. One way to avoid this idling is to force all the processors to spend the same time in the preconditioning application in each outer iteration. The rationale is that it is better to spend the time that would otherwise be wasted, to perform more iterations in the "faster" processors. A better accuracy may be achieved in these processors which would eventually propagate to others, resulting in a reduction of the number of iterations. The time may be fixed, for example, based on the time required by the processor with the largest workload to apply a preconditioning step.

There are several approaches to control the "fixed-time" condition for an Additive Schwarz application which uses an iterative process (say, preconditioned GMRES). One of these approaches is to change the number of inner iterations at

a certain iteration of FGMRES for each processor based on some criterion comparing the time of the previous preconditioning step. Thus the processors with small workloads will proceed for more iterations and compute a more accurate preconditioning vector. As a result, the number of the outer iterations may be reduced.

In testing this approach, the following iteration adjustment has been determined experimentally and applied after each preconditioning step in processor i, $(i = 1, \ldots, p)$:

$$\text{if } (\Delta_j^i > n_{j-1}^i/3) \quad n_j^i = n_{j-1}^i + \Delta_j^i,$$

where n_j^i is the number of the inner iterations in the jth iteration of FGMRES; Δ_j^i is the number of iterations that processor i can fit into the time to be wasted in idling otherwise at the (next) jth outer iteration of FGMRES. Specifically,

$$\Delta_j^i = \frac{(T_{\max} - T^i)N^i}{T^i},$$

where $T_{\max}$ is the maximum time among all the processors and T^i the time for processor i to perform preconditioning operations during $j-1$ previous outer iterations, N^i is the total number of preconditioning operations performed by processor i so far. The number of inner iterations n_j^i can be updated provided that the limit n_{lim} on the number of inner iterations is not reached. Figure 6 compares the time and iteration results for the standard (`Jacobi` in Figure 6) and the "fixed-time" (`Jacobi_ft` in Figure 6) Additive Schwatz preconditionings accelerated with FGMRES(20). The experiments have been performed on the IBM SP with the problem AF23560 from the Harwell-Boeing collection [2]. This problem has 23,560 unknowns and 484,256 nonzeros in the matrix. The preconditioning phase consists of ILUT-preconditioned GMRES(20) with the following parameters: `lfil`=25, $n_0^i = 5$, $n_{\text{lim}} = 20$, and the relative accuracy of 10^{-2}. Note that this accuracy is increased to the accuracy of FGMRES (10^{-8}) whenever n^i is increased for the "fixed-time" algorithm. Figure 6 indicates that `Jacobi_ft` exhibits a better overall performance on a wide range of processor numbers, thus showing an advantage of forced load balancing. Earlier testings of this approach with a slightly different criterion, reported in [11] for small processor numbers, are in agreement with our numerical experiments.

4 Conclusion

A few strategies have been described for enhancing the performance of preconditioned Krylov subspace methods for solving distributed sparse linear systems. The techniques described show that much can be gained from using partitioners that take into account the values of the matrix entries. In addition, the standard stopping criteria used when iterating locally in a processor may lead to wasted idle time, either because of load imbalance or because some subdomain systems are much easier to solve than others. We found that some improvements can be made by simply forcing all processors to iterate about the same amount of time.

Such non-standard heuristics may be necessary because the impact of a given partitioning on the overall preconditioner is difficult to predict or analyze.

References

1. X. C. Cai and Y. Saad. Overlapping domain decomposition algorithms for general sparse matrices. *Numerical Linear Algebra with Applications*, 3:221–237, 1996.
2. I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15:1–14, 1989.
3. V. Eijkhout and T. Chan. ParPre a parallel preconditioners package, reference manual for version 2.0.17. Technical Report CAM Report 97-24, UCLA, 1997.
4. C. Farhat and M. Lesoinne. Mesh partitioning algorithms for the parallel solution of partial differential equations. *Applied Numerical Mathematics*, 12, 1993.
5. J. A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
6. T. Goehring and Y. Saad. Heuristic algorithms for automatic graph partitioning. Technical Report UMSI 94-29, University of Minnesota Supercomputer Institute, Minneapolis, MN, February 1994.
7. B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report SAND92-1460, UC-405, Sandia National Laboratories, Albuquerque, NM, 1992.
8. Scott A. Hutchinson, John N. Shadid, and R. S. Tuminaro. Aztec user's guide. version 1.0. Technical Report SAND95-1559, Sandia National Laboratories, Albuquerque, NM, 1995.
9. M. T. Jones and P. E. Plassmann. BlockSolve95 users manual: Scalable library software for the solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Lab., Argonne, IL., 1995.
10. G. Karypis. *Graph Partitioning and its Applications to Scientific Computing*. PhD thesis, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1996.
11. S. Kuznetsov, G. C. Lo, and Y. Saad. Parallel solution of general sparse linear systems. Technical Report UMSI 97/98, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1997.
12. J. W. H. Liu. A graph partitioning algorithm by node separators. *ACM Transactions on Mathematical Software*, 15:198–219, 1989.
13. A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11:430–452, 1990.
14. Y. Saad. Parallel sparse matrix library (P_SPARSLIB): The iterative solvers module. In *Advances in Numerical Methods for Large Sparse Sets of Linear Equations, Number 10, Matrix Analysis and Parallel Computing, PCG 94*, pages 263–276, Keio University, Yokohama, Japan, 1994.
15. Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS publishing, New York, 1996.
16. Y. Saad and A. Malevsky. PSPARSLIB: A portable library of distributed memory sparse iterative solvers. In V. E. Malyshkin et al., editor, *Proceedings of Parallel Computing Technologies (PaCT-95), 3-rd international conference, St. Petersburg, Russia, Sept. 1995*, 1995.

17. Y. Saad and M. Sosonkina. Distributed Schur complement techniques for general sparse linear systems. Technical Report UMSI 97/159, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1997. Submitted, Revised.
18. B. Smith, P. Bjørstad, and W. Gropp. *Domain decomposition: Parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, New-York, NY, 1996.
19. B. Smith, W. D. Gropp, and L. C. McInnes. PETSc 2.0 user's manual. Technical Report ANL-95/11, Argonne National Laboratory, Argonne, IL, July 1995.

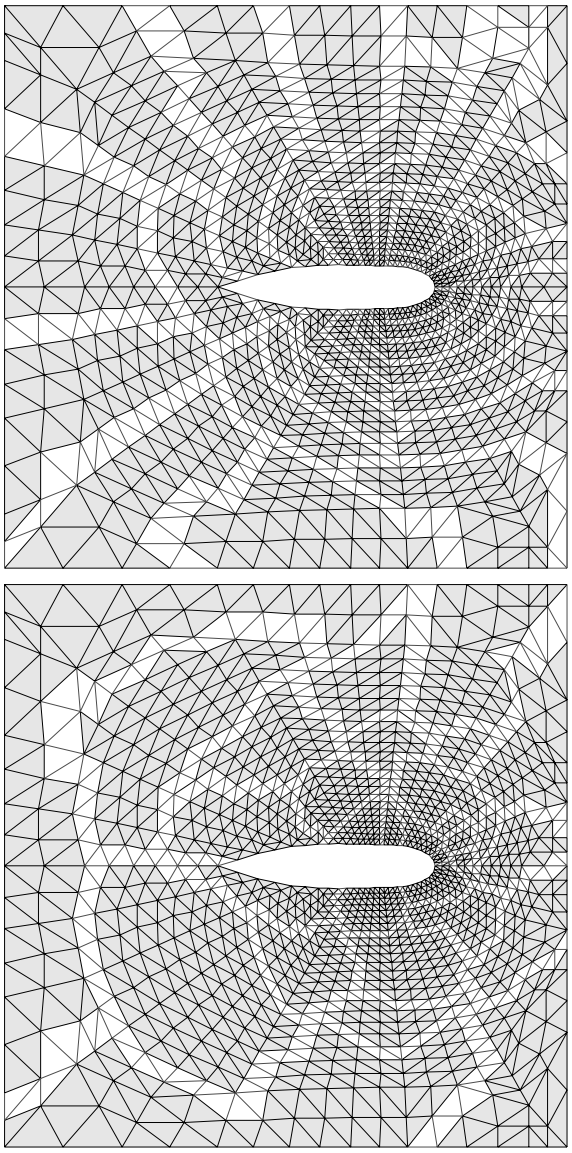

Fig. 2. Two-way partitioning (left) and level-set expansion using centers at the "middle node" of resulting subdomains (right).

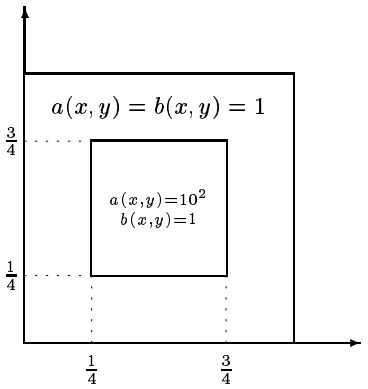

Fig. 3. A test problem with discontinuous coefficient.

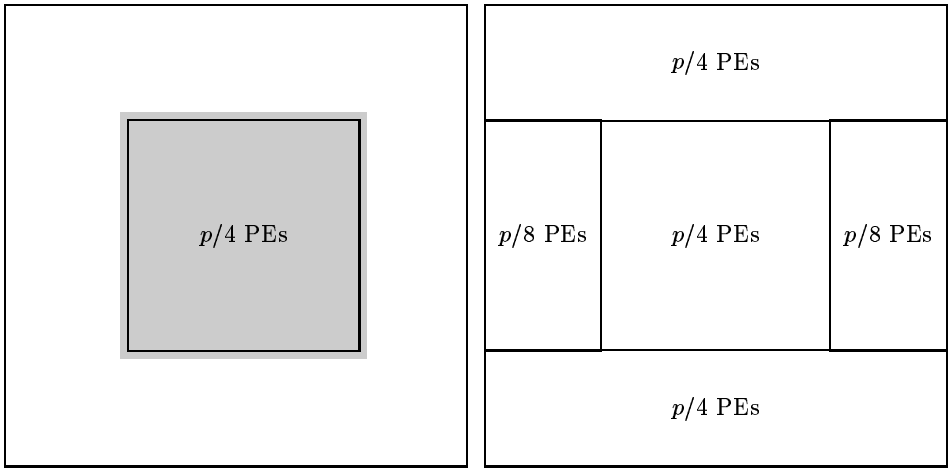

Fig. 4. A macro-partitioning of the problem in Figure 3.

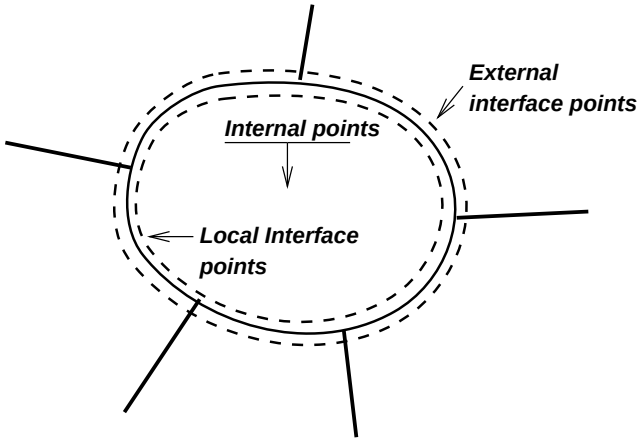

Fig. 5. A local view of a distributed sparse matrix.

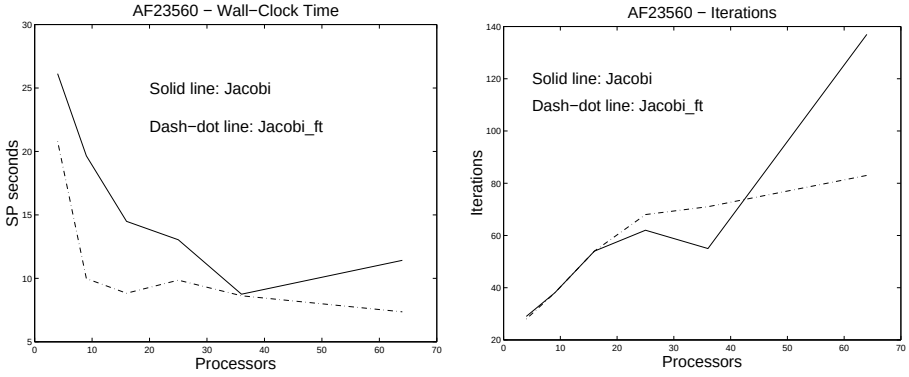

Fig. 6. Solution times and iterations for the standard and "fixed-time" Additive Schwatz preconditionings.